Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Planificación en sistemas robotizados mediante PDDL y ROS.

Autor: Javier Ortiz Perez-Jaraiz

Tutor/es: Ángel Javier Álvarez Miguel

2021

Agradecimientos

> No estoy seguro de que yo exista, en realidad. Soy todos los autores que he leído, toda la gente que he conocido, todas las mujeres que he amado. Todas las ciudades que he visitado, todos mis antepasados.
>
> *Jorge Luis Borges*

Siempre he estado muy de acuerdo con Jorge Luis Borges en la discusión sobre la identidad y la existencia, no creo en el hombre hecho a sí mismo, ni me gusta pensar que estamos predestinados a ser lo que somos, como decían en una de mis películas favoritas "No me gusta la idea de no ser yo quien maneje mi propia vida", tampoco me parece extraño que la sociedad se comporte como un sistema abierto, donde las interacciones del entorno conformen a las personas y las personas influyan al entorno. Me gustaría utilizar este apartado para agradecer a todas aquellas personas que han pasado por mi vida y de las que alguna manera conforma las persona que soy, espero que en algún sentido estéis orgullosos de en lo que me he convertido.

Hay muchas personas que han pasado por mi vida y más en estos cuatro años de carrera universitaria, algunas estaban antes de empezar y ya no están, otras estuvieron durante el proceso pero en algún momento separamos caminos, unas pocas las conocí durante el camino y siguen estando, y otras pocas estaban antes y aún siguen conmigo, igualmente este trabajo no lo habría escrito sin todas ellas. A continuación, me gustaría prestar más detalle a aquellas que me han marcado más.

A mi familia por estar siempre a mi lado y haberme dado todo lo que necesitaba, especialmente a mi abuela y a mi madre. A mi abuela por haberme permitido quedarme en su casa durante estos últimos seis años, permitiendo así que estudiase lo que quería y donde quería, espero que podamos estar mucho más tiempo juntos. A mi madre por siempre haber creído en mi incluso cuando nadie más lo hacía, no creo que exista una mayor expresión de amor.

A mis amigos de toda la vida, Elvira, David, Kike, Iván y Costi, me han visto cambiar durante estos cuatros años y yo los he visto cambiar a ellos, queda poco de lo que nos unía en antaño, cada uno ha empezado a recorrer su propio camino y no nos vemos tanto como antes, pero lo que podría haber sido un periodo donde nuestra relación se enfriara y se rompiera, se ha convertido en un periodo donde se han vuelto mi principal apoyo, sin ninguna duda estoy muy orgulloso de ellos.

A mis amigos de la universidad, admito que cuando entre en la universidad me encontraba un poco asustado, la EPS era un edificio abrumador, con una biblioteca llena de libros que contenía un montón de cosas que siempre había querido aprender, brazos robóticos y ordenadores por los pasillos, no sabía si podría dar la talla o si iba a ser mi lugar. Mis dudas duraron poco cuando conocí a César y a Javi Quintanar en la charla de bienvenida, no sé cómo pudo ser tan sencillo, parecía que éramos amigos de toda la vida, con los mismos chistes, el mismo humor, los mimos intereses y los mismo miedos. Hemos trabajado juntos estos cuatro años, en las buenas y en las malas, y tengo claro que no podría haber llegado hasta aquí sin esa simbiosis que tanto nos caracteriza trabajando.

Al proyecto de Eurobot Senior o el UAH Robotics, se que es un poco extraño meter un proyecto docente en medio de personas tan importantes en mi vida, pero al fin y al cabo me estoy graduando de una ingeniería y quizás he pasado incluso más hora de las debidas en este proyecto. Lo que fue una idea tonta de hacer un robot entre Javi, César y yo porque nos aburríamos en verano, se convirtió en uno de los lugares donde más he aprendido en toda mi vida, habrán sido tragos más dulces o más amargos pero en todos ellos he aprendido algo. Así que sin duda estoy agradecido con las escuela politécnica superior y a la universidad por haber creado este proyecto y haberlo continuado durante estos años.

A mis dos maestros Javier Guerreo y Ángel Javier Álvarez, dos personas que admiro y se han convertido en mis mayores referentes a seguir. Desde el momento en el que los conoció han intentado que sea la mejor persona posible sin haber dado yo nada a cambio, nunca han dudado en corregirme y siempre han mirado por mi bien por encima del suyo, siendo sin duda personas de buen corazón y de pequeños gestos más que de grandes gestas. Sobre todo, me gustaría destacar la perseverancia y la actitud de Javi, valores que no solo me ha transmitido a través de las artes marciales, sino que sin ninguna duda portaba dentro y fuera del tatami. De Ángel destacaría las ganas de aprender y perder el miedo a fallar, cuando lo conocí en primero de carrera en un Codenares Lite y empezamos a hablar solo entendía un 20 % de lo que me quería transmitir o de lo que estaba hablando, con el paso del tiempo creo que he llegado a tener alguna conversación donde he entendido todo lo que me quería decir, pero sin ninguna duda sigue habiendo muchos días de los que hablamos en los que me quedo despierto hasta las tantas, buscando por internet el tema del que hemos hablado para que ya no me pille por sorpresa la próxima vez que salga

Resumen

La planificación automática se encarga de generar un plan con las distintas acciones que logran cumplir los objetivos definidos. Esta habilidad resulta útil en sistemas robotizados, porque permite definir el comportamiento de uno o varios robots sin tener que predefinir las acciones que consiguen cumplir un objetivo. En este trabajo presentaremos las principales herramientas que se pueden utilizar para integrar sistemas de planificación automática en entornos robotizados mediante el uso de ROS. Además, estas herramientas se integrarán en sistemas robóticos reales mediante el uso del simulador Gazebo, lo que permitirá una mayor comprensión sobre el funcionamiento del sistema.

Palabras clave— IA, Robots, ROS, Planificación automática, PDDL.

Abstract

Automatic planning is responsible for generating a plan with the different actions that achieve the defined objectives. This ability is useful in robotic systems, because it allows defining the behavior of one or more robots without having to predefine the actions that achieve a goal. In this work we will present the main tools that can be used to integrate automatic planning systems in robotic environments through the use of ROS. In addition, these tools will be integrated into real robotic systems using the Gazebo simulator, which will allow a greater understanding of the system.

Keywords— AI, Robots, ROS, Automated Planning, PDDL.

Índice

Índice de figuras

Índice de tablas

1. Introducción y estructura

1.1. Introducción

El presente trabajo fin de grado tiene como principal objetivo el estudio de las técnicas de planificación automática en entornos robotizados y su integración. Para apoyar la explicación de este trabajo se establece como objetivo secundario integrar estas herramientas en un entorno de simulación para facilitar así la comprensión sobre el sistema desarrollado.

Este trabajo pertenece a dos de los campos con más auge en la investigación, la robótica y la inteligencia artificial. Estos dos campos campos llevan siendo de gran interés tanto para empresas como para investigadores desde hace más de medio siglo, ya que permiten tanto la automatización de acciones mecánicas, en el caso de la robótica, como la automatización de toma de decisiones que maximicen las probabilidades de tener éxito en algún objetivo o tarea, en el caso de la inteligencia artificial.

También es inmensa la cantidad de trabajos que, al igual que el aquí presente, se encuentran enmarcados en estos dos campos. Siendo últimamente muy comunes los trabajos que tratan de adaptar técnicas de Inteligencia Artificial como el Machine Learnig a sistemas robotizados. Al contrario, este trabajo busca la adaptación de algoritmos de inteligencia artificiales "clásicos" para definir el comportamiento de estos sistemas. Este enfoque resulta interesante porque a diferencia de otras herramientas típicas de diseño de comportamiento de robots, como los arboles de comportamiento, las maquinas de estado finitas y el comportamiento reactivo, la planificación permite definir el comportamiento del sistema sin tener que indicar los pasos para llegar el objetivo, limitándonos a definir el estado del mundo, las acciones que puede realizar el robot y el estado final que se desea alcanzar.

Para poder adaptar este modelo de comportamiento a un sistema real como es el caso de un robot y convertirlo en un agente inteligente, usaremos la arquitectura descrita en la figura 1, siendo una de las arquitecturas más comunes a la hora de adaptar sistemas de planificación a sistemas robotizados.

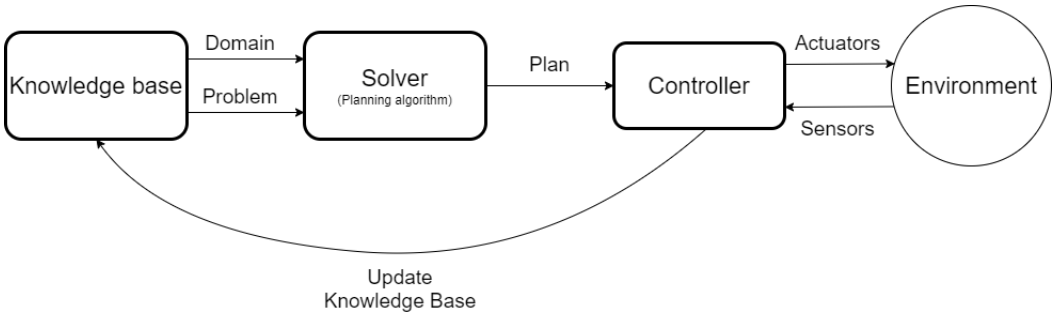

Figura 1: Arquitectura del agente inteligente [23]

En esta figura podemos encontrar los siguientes elementos:

- Base de conocimientos (Knowledge Base): Contiene toda la información sobre las acciones que puede realizar el robot, la tarea a resolver y la información disponible del entorno. Esta información estará codificada mediante el lenguaje PDDL (Planning Domain Definition Language), siendo PDDL el lenguaje estándar adoptado por la International Planning Competition (IPC) para definir problemas de planifi-

cación clásicos. En PDDL esta información se encuentra repartida en dos archivos, el fichero Domain y el fichero Problem.

- Planificador (Solver): Es el encargado de leer el problema de planificación que se encuentra en los ficheros Domain y Problem de la base de conocimiento, y resolverlo mediante un algoritmo de planificación. La solución a este problema será un plan en el que se indique las acciones que debe que realizar el robot para resolver la tarea.
- Controlador (Controller): Se encarga de ejecutar el plan mediante los actuadores del robot y de actualizar la información del entorno en la base de conocimiento gracias a los sensores del robot.
- Entorno (Enviroment): Es el conjunto de circunstancias que definen la situación de nuestro agente.

Todo este estudio se realizara en ROS (Sistema operativo robótico), este framework nos permitirá abstraer el hardware de los distintos robots, separar los distintos componentes de la arquitectura del agente en diferentes hilos de procesador y nos facilitará el desarrollo de las herramientas de visualización.

1.2. Estructura del TFG

Este trabajo se estructura en una serie de capítulos, esto nos permite encapsular cada una de las ideas trabajadas, mejorando la legibilidad del trabajo. Las ideas que se van a desarrollar en cada uno de los capítulos son las siguiente:

- Capítulo 1: Contiene un breve resumen del contenido del trabajo, su objetivo y su estructura.
- Capítulo 2: Contiene el estado del arte relativo a la planificación automática y a ROS. Además de explicar los conceptos clave de ROS y PDDL.
- Capítulo 3: Contiene un análisis de las distintas herramientas existentes que permiten la integración directa de Planificadores en ROS y en ROS2.
- Capítulo 4: Contiene el desarrollo y la integración de un sistema de planificación en un entorno robotizado utilizando ROS. Este capitulo servirá para captar la forma de trabajar en ROS, además de aprender a representar un problema de planificación mediante PDDL. En este capitulo se creará un entorno de simulación utilizando el simulador gazebo para ver el completo funcionamiento del sistema.
- Capítulo 5: Contiene el desarrollo y la integración de un sistema de planificación en un entorno robotizado utilizando ROS2. En este capitulo se profundizará sobre algunas extensiones y comportamientos útil de PDDL en el mundo de la robótica, además de comentar las ventajas que nos ofrece la nueva versión del framework robótico.
- Capítulo 6: Contiene el desarrollo y la integración de un sistema de planificación en la arquitectura software de los robots del UAH Robotics Team que participaron en la competición Eurobot 2021. En este capitulo se verá como se represento el dominio para esta competición y como adaptar la estructura de planificación a un sistema robotizado de competición.
- Capítulo 7: Contiene las conclusiones finales y futuros trabajos basados en este TFG.

2. Estado del arte

2.1. Robótica e inteligencia artificial

La robótica es una rama que estudia el proceso de construir maquinas programables, habitualmente de carácter electromecánico, capaces de realizar tareas complejas de manera automática. Por otro lado la inteligencia artificial es definida en algunos textos, como una rama de las ciencias de la computación que estudia a los agentes inteligentes, definiendo una agente inteligente como cualquier sistema que perciba su entorno y lleve a cabo acciones que maximicen sus posibilidades de lograr sus objetivos, como pueden ser los animales, los humanos o las plantas. No es difícil ver que entre estas dos ramas existe una relación simbiótica que permite por un lado, mejorar la capacidad de la robótica para realizar acciones de forma automática y autónoma, y por otro lado dotar a los agentes inteligentes de una estructura mecánica con la que interactuar con el entorno, y una interfaz sensorial con el que percibirlo.

Esta relación se puede considerar que comienza en 1948 con la creación de Elmer y Elsie los primeros robots autónomos. Estos dos robots con forma de tortuga fueron creados por el neurobiólogo William Grey Walter, el diseño contaba con dos motores para el desplazamiento e intentaba emular la conexión entre dos células nerviosas utilizando una célula fotoeléctrica y un sensor de contacto, consiguiendo un comportamiento parecido a la fotaxia de las células, es decir la capacidad que tienen las células de realizar movimientos en dirección a los estímulos luminosos. Este neurobiólogo consiguió dotar de cuatro comportamientos a estos dos robots un patrón de exploración , fototaxia, fotofobia o fotaxia negativa y un patrón para esquivar obstáculos.

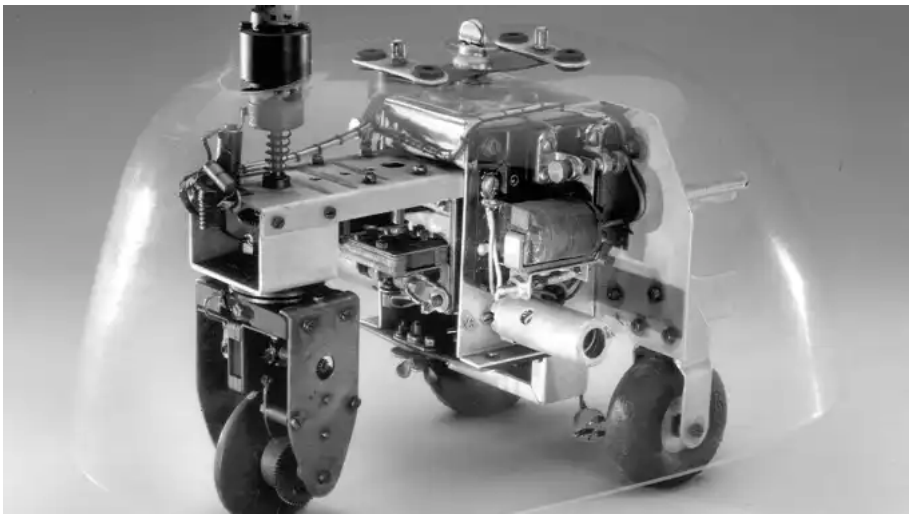

Figura 2: Estructura de los robots Elmer y Elsie

Otro de los grandes hitos de la combinación entre estas dos ramas es el robot Shakey creado por la DARPA a finales de los 60. Shakey es considerado el primer robot de propósito general inteligente, este a diferencia de sus predecesores era capaz de analizar las distintas tareas y dividirlas formando instrucciones más pequeñas, en cambio sus predecesores estaban programados de ante mano para saber las instrucciones que conseguían cumplir el objetivo. Es por esta diferencia por lo que se considera que Shakey era consciente de sus acciones y por ello inteligente.

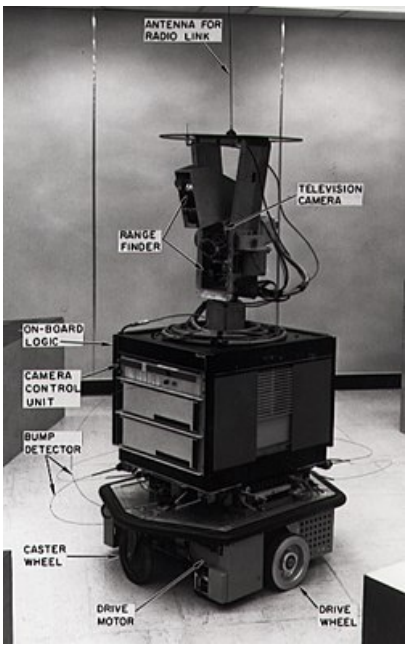

Figura 3: Robot Shakey

A medida que pasaron los años tanto la inteligencia artificial como la robótica fueron desarrollándose creando nuevas técnicas y nuevos modelos, entre los mayores avances destacan el desarrollo en los sistemas expertos logrado en los años 80, las mejoras en las técnicas de percepción y en los sistemas de planificación. Consiguiendo grandes hitos como el Stanford Cart, el primer vehículo autónomo, el Sojourner rover, el primer robot autónomo desplegado en Marte, o el robot Asimo, uno de los humanoides más avanzado. En la actualidad casi todos los robots presentan comportamientos inteligentes, siendo cada vez más abundantes los robots que integran técnicas de aprendizaje automático.

2.2. Planificación automática

La planificación automática es un área de inteligencia artificial en la que se estudia los efectos de una secuencia de acciones, en palabras de Ghallab et al en el libro Automated Planning: Theory and Practice [15], "Planning is the reasoning side of acting. It is an abstract, explicit deliberation process thtat chooses and organizes actions by anticipating ther expected outcomes. This deliberation aims at achieving as best as possible some pre stated objectives. Automated plannning is an area of Artificial intelligence (AI) that studeis this delibration process computationally".

Gran parte de la rama de planificación automática se centra en resolver problemas de planificación mediante el uso de computadores, los problemas de planificación consisten en encontrar una secuencia de acciones, es decir un plan, con las que transicionar desde un estado inicial hasta un estado final donde se cumplen los objetivos del problema. Esta resolución de problemas de planificación de manera automática, tiene un alto valor en sectores como la logística, el montaje de maquinaria, la organización de proyectos complejos y la robótica.

2.2.1. Planificación clásica

Dentro de los problemas de planificación se establecen una serie de modelos que se caracterizan por el número de restricciones o suposiciones que se hacen acerca del problema. A continuación se presentan las restricciones más comunes dentro de los problemas de planificación:

- Mundo finito o Infinito: En el mundo finito el entorno puede encontrase en un número finito de estados, en cambio en el infinito hay infinitas posibilidades.
- Observabilidad total o parcial: Si el agente tiene acceso a toda la información del estado actual del entorno se dice que existe una observabilidad total, en cambio si hay hechos que son desconocidos se dice que hay una observabilidad parcial.
- Mundo determinista o estocástico: En un mundo determinista no existen efectos estocásticos, por lo que las acciones siempre producen el mismo efecto sobre el estado. En cambio en un mundo estocástico las acciones tienen cierta probabilidad de causar algunos efectos.
- Mundo estático o dinámico: En un mundo estático el estado del problema solo se ve modificado por las acciones del agente que realiza el plan, en un mundo dinámico el entorno se puede ver modificado por agentes externos o eventos exógenos.
- Restricciones intermedias o sin restricciones intermedias: Cuando en un problema existen restricciones intermedias los estados intermedios, aquellos que se encuentran entre el estado final y el inicial, tienen que cumplir una serie de condiciones. En cambio si no existen restricciones intermedias el único estado con restricciones es el final, en el que se tienen que cumplir los objetivos.
- Plan secuencial o concurrente: Cuando un agente solo puede ejecutar acciones de una en una y secuencialmente nos encontramos ante un plan secuencial. Si el agente puede realizar varias acciones en un mismo instante de tiempo estamos ante un plan concurrente.
- Tiempo implícito o explicito: Si se considera que todas las acciones tienen el mismo coste de tiempo o que su duración es nula se dice que el tiempo es explicito. En cambio si el planificador tiene en cuenta el tiempo que tarda en realizarse cada acción el tiempo es implícito.
- Planificación offline o Online: La planificación online tiene en cuenta la dinámica del entorno, por lo que el plan se va presentando a medida que se realizan las acciones ya que las observaciones pueden verse modificadas. En la planificación offline estas dos fases no se ven intercaladas, primero se halla el plan completo y luego se ejecuta.

La planificación clásica es un modelo bastante restrictivo que supuso un punto de partida desde el que abordar los problemas de la planificación automática, cuenta con las siguientes restricciones: mundo finito, observabilidad total, mundo determinista, mundo estático, sin restricciones en estados intermedios, planes secuenciales, tiempo implícito y planificación offline.

2.2.2. Formalización proposicional

Para representar los problemas de planificación clásica existen dos formalizaciones principales, la formalización proposicional y la formalización multivaluada. Puesto que el lenguaje que se va a utilizar mayoritariamente en este trabajo fin de grado es PDDL, que utiliza la formalización proposicional, pasaremos a describirlo a continuación. Tal y como indica Nerea Luis Mingueza en su Tesis Plan Merging via Plan Reuse [23], en la formalización proposicional aparecen los siguientes elementos:

El problema que se describe como la siguiente tupla, $\Pi = \langle F, A, I.G \rangle$ donde:

- F es el conjunto de proposiciones.
- A es el conjunto de acciones.
- I representa el estado inicial.
- G es el conjunto de objetivos.

  Tanto el estado inicial I como el conjunto de objetivos G están contenidos en el conjunto de proposiciones.

Un estado s es un conjunto de valores de F instanciados, los cuales describen la situación del problema. A su vez el conjunto formado por todos los posibles estados es el espacio de estados S, por lo que $I \in S$, este conjunto en la planificación clásica es finito y discreto.

A su vez las acciones se describen como otra tupla, siendo a una acción $a \in A$, se puede expresar $a = \langle Pre(a), Eff(a), C(a) \rangle$ donde:

- Pre es el conjunto de precondiciones de la acción a, son literales que deben ser ciertos para poder aplicar la acción a sobre un estado.
- Eff es el conjunto de los efectos de la acción a, y son los literales que van a ser modificados del estado tras la aplicación de la acción a. En la modificación se pueden distinguir dos conjuntos dependiendo de su efecto $add(a)$ que son los literales que se añaden al estado y $del(a)$ los literales que se borran del estado.
- C es un número positivo que representa el coste de la acción, por defecto el coste de la acción se considera uno.

Y la función que expresa la aplicación de una acción a sobre un estado s para vienen definida por la función de transición γ. Donde γ es:

$$\gamma(s,a) = \begin{cases} ((s \backslash del(a)) \cup add(a) & si \quad pre(a) \subseteq s \\ \\ s & si \quad pre(a) \not\subseteq s \end{cases}$$

La solución de un problema de planificación es un plan que consisten en una secuencia de acciones $\pi = (a_1, ..., a_n)$ que si son ejecutadas partiendo del estado inicial I se consigue alcanzar el estado final G, satisfaciendo $G \subseteq S_g$, donde S_g es el estado final. Pudiendo representarse la ejecución del plan π como $Gamma$ siendo:

$$\Gamma(s,a) = \begin{cases} (\Gamma(\gamma(s,a_1)(a_2, ...a_n))) & si \quad \pi \neq \emptyset \\ \\ s & si \quad \pi = \emptyset \end{cases}$$

2.2.3. Algoritmos de búsqueda

Sabiendo que es un problema de planificación y como representarlo de manera proposicional, faltaría explicar la manera de resolverlo. Hay numerosos algoritmos que se pueden aplicar para resolver estos problemas, la técnica más común es construir un grafo con el estado inicial, y aplicar un algoritmo de búsqueda a la vez que se van explorando los nodos. Una forma bastante común de clasificar estos algoritmos es dependiendo del sentido en el que se exploren los estados, existiendo tres sentidos tal y como nos indica el libro Automated Planning: Theory and Practice [15].

- Forward Search: Son los algoritmos más comunes, inician su búsqueda desde el estado inicial hasta el estado final construyendo una secuencia de acciones, como la búsqueda empieza en el estado inicial y para construir los estados se han de cumplir las precondiciones de las acciones, todos los estados están completamente definidos y son alcanzables. Ejemplos de este tipo de algoritmos son la búsqueda en profundidad, el A*, el AO* y el IDA*, estos dos últimos son utilizados en los planificadores Contigent-FF y el Metric-FF respectivamente.
- Backward Search: Inicia su búsqueda en el estado final y avanza hacia el estado inicial. En este tipo de búsqueda los estados esta parcialmente definidos, puesto que del estado final solo se conocen las condiciones objetivo, esto provoca que el resto de condiciones sean desconocidas y haya multitud de estados completos que puedan cumplirlas. El HSPr era uno de los planificadores que utilizaban estos algoritmos.
- Bidirectional Search: Consiste en una mezcla entre los dos tipos anteriores, este tipo de algoritmo realiza la exploración en ambos sentidos encontrando la solución en la frontera.

2.2.4. Planificadores y lenguajes

Para poder resolver problemas de planificación mediante un ordenador son necesarias dos cosas, un lenguaje descriptivo con el que poder expresar los distintos problemas, también conocido como lenguaje de acción , y un programa que lea el problema expresado en el alguno de estos lenguajes y lo resuelva, a estos programas se les denomina planificadores o solvers. Desde mediados del siglo XX se han desarrollado numerosos lenguajes de acción y planificadores, a continuación se presentan los más importantes:

- GPS: El General Problem Solver fue el primer programa de planificación general, este programa nació en 1957 y tenía como objetivo resolver problemas de carácter general entre los que se encontraban problemas de planificación clásica, para poder resolver este tipo de problemas utilizaba la demostración de teoremas y la lógica de predicados.
- STRIPS: En 1971 se creo el STanford Research Institute Problem Solver también conocido como STRIPS. Se conoce por STRIPS tanto al planificador automático como al lenguaje descriptivo que utilizaba el solver. Fue el primero en emplear la formalización proposicional, y aunque STRIPS esta ideado para para problemas de planificación clásicos, han ido surgiendo otras variedades del lenguaje para enfrentarse a dominios menos restrictivos como MA-STRIPS (Multi agent strips) enfocado en la planificación multiagente.

- HTN: EL Hierarchical task network es un lenguaje de acción surgido en la década de los 70 en el que el problemas de planificación se describen como una red jerárquica de tareas. En esta red se distinguen varios niveles de complejidad, las tareas primitivas que corresponden con las acciones de la proposición formal, las tareas intermedias que son un conjunto de primitivas, y las tareas objetivo que se parecen a los objetivos de la formalización proposicional pero son aun más generales. La ventaja de este sistema es que es bastante más expresivo que lenguajes de acción como STRIPS, la desventaja es que la complejidad de estos aumenta. Algunos de los planificadores que utilizan este lenguaje son Nonlin, SIPE-2, O-Plan, UMCP y SHOP2.

- POP: Los partial-order planning o planificadores de menor compromiso son planificadores de orden parcial, su principal característica es que ciertas acciones del plan no se encuentran ordenadas, esto es debido a que este tipo de planificadores dividen los objetivos en distintos subobjetivos, y realizan varios subplanes con los que alcanzar cada uno de estos subojetivos, para luego juntarlos en un mismo plan. Algunos ejemplos de planificadores parciales son el UCPOP Y el SNLP.

- SATPLAN: Es unas técnica que se empezó a desarrollar en 1992 y permite transformar los problemas de planificación en problemas de satisfabilidad booleana. La ventaja que produce es que los problemas de satisfabilidad booleana están muy estudiados, por lo que existen muchas técnicas ya desarrolladas para resolverlos como son DPLL O WalkSAT. Un ejemplo es el planificador Madagascar.

- Graphplan: Graphplan fue creado en 1995 por Avrim Blum y Merrick Furst, es un planificador que recoge un problema escrito en STRIPS y produce un plan mediante la exploración de un grafo de planificación. Normalmente los planificadores hacen una exploración de un grafo de espacio de estados, en este tipo de grafo los nodos son posibles estados y los enlaces son las acciones para transicionar de un estado a otro, en cambio en los grafos de planificación cada nodo es un conjunto de operadores parcialmente instanciados, más algunas restricciones. Concretamente el grafo de planificación que utiliza Graphplan se construye realizando capas de nodos, existen dos tipos de capas, capas cuyos nodos son proposiciones atómicas y capas cuyos nodos son acciones, estas capas se enlazan mediante precondiciones y efectos de acción. Esta técnica permite reducir el espacio de búsqueda ya que tiene un factor de ramificación mucho menor a pesar de generar más nodos. Es muy común ver en planificadores modernos realizar esta técnica como prefiltrado para luego aplicar algoritmos de búsqueda marcha atrás, un ejemplo es el sover AltAlt que mezcla el grafo de planificación de Graphplan con el algoritmo de búsqueda de HSP-r.

- ADL: El Action description language es un lenguaje de acción que supone una extensión sobre STRIPS, nace en 1997 y esta enfocada en la programación de sistemas robotizados. Entre las mejoras introducidas por ADL esta la posibilidad de que un operador pueda tener efectos condicionales, que las acciones puedan ser descritas mediante efectos indirectos, permite utilizar la condición de igualdad y las proposiciones pueden estar clasificadas mediante leyes estáticas y dinámicas.

- HSP: Los Heuristic Saearch Planners son planificadores que convierten el problema de planificación en un problema de búsqueda que resuelven utilizando heurísticas independientes del dominio. Entre las más utilizadas están la heurística aditiva, la heurística de máximo coste, la relajación de restricciones o la heurística basada en puntos de referencia. Actualmente la gran mayoría de planificadores utilizan heurísticas independientes del dominio, como la familia de solvers FF o la HSP.

2.3. PDDL

PDDL son las siglas de planification domain description language, es una familia de lenguajes de acción que permite representar problemas de planificación mediante la formalización proposicional. El primer manual que recogía su sintaxis fue publicado en 1998 baja el nombre PDDL - The Planning Domain Definition Language Version 1.2 [17], los autores de este documento son Malik Ghallab, Drew McDemort y el comité internacional del concurso de planificación (IPC). En aquel entonces en la comunidad de planificación automatizada había una necesidad de poder comparar empíricamente los planificadores automáticos, por esta razón se creo la Artificial Intelligence Planning and Scheduling Competition (AIPS), actualmente la International Conference on Automated Planning and Scheduling (ICAPS), no bastaba con crear solo el organismo si no que era necesario crear un lenguaje de acción común con el que expresar los problemas de planificación, ya que había muchos lenguajes presentes en la comunidad como STRIPS, ADL o HTN. Fue entonces cuando se desarrollo PDDL con la intención de convertirse en el estándar de facto para representación de los problemas de planificación. Citando a los desarrolladores de PDDL "Our hope is to encourage sharing of problems and algorithms, as well as to allow meaningful comparison of the performance of planners on different problems".

PDDL fue altamente influenciado por la expresividad de ADL y la expresividad de acciones de UMCP, aunque también tienen otras influencias como SIPE-2, Prodigy-4.0, Unpop y UCPOP. PDDL utiliza la representación proposicional e intenta tener una estructura modular que permita ir añadiendo más expresividad y nuevas funcionalidades sin necesidad de tocar la sintaxis del lenguaje, y sin que tampoco sea necesario crear un nuevo lenguaje de acción para implementarla. Igualmente años más tarde se crearan otros lenguajes dentro de la familia de PDDL que hereden parte de su sintaxis para poder modelar problemas alejados de la planificación clásica, como pueden ser PPDDL, NDDL, RDDL y MADDL.

2.3.1. Representación mediante PDDL

Para codificar un problema de planificación utilizando PDDL son necesarios dos ficheros, uno de ellos es el fichero domain que contiene la información acerca del dominio, desde el tipo de objetos que pueden llegar a existir en el mundo, como las proposiciones que puede haber o las acciones que puede realizar el agente. Por el otro lado esta el fichero problem que contiene la información detallada de los objetivos del problema, los objetos que existen en el mundo y la situación inicial del problema. Esta separación permite crear múltiples problemas de planificación para un único dominio. A continuación se describen las secciones en las que se puede dividir el fichero dominio:

- Meta-información del problema: Se encuentra al principio del fichero y en ella se define el nombre del dominio y los requerimientos del problema. El nombre del dominio permite identificar a que dominio pertenece un determinado problema, por otro lado los requerimientos indican características que se van a utilizar del lenguaje. Los requerimientos permiten que PDDL sea un lenguaje modular, puesto que con ellos se pueden incluir extensiones de las distintas versiones de PDDL sin modificar la sintaxis original. Que un planificador soporte todos los requerimientos no solo no es obligatorio, si no que tampoco es lo habitual, en su diseño la IPC sabía que no todos los planificadores iban a ser capaces de soportar la sintaxis que se fuera añadiendo a PDDL con el paso de los años.

- Tipos: Esta sección nos permite crear una jerarquía de tipos, en la que pueden existen tipos básicos que no heredan de ningún otro tipo y tipos más complejos que heredan de un único tipo. Esto nos permite crear condiciones que afecten a muchos objetos si se emplea en un tipo básico o condiciones más especificas si esta en un nivel más alto de la jerarquía.

- Predicados: En esta parte se definen los tipos de valores proposicionales que puedo haber en el dominio, estos valores son atómicos y bivalentes , es decir están sometidos a un valor de verdadero o falso, y pueden ser propios de un único objeto o pueden definirse entre varios objetos. La instancia de todos los predicados para un posible problema conforman el conjunto de proposiciones es decir F.

- Acciones: Cada una de las acciones que puede realizar el agente se define de forma individual, cada acción tiene un bloque de parámetros, un bloque de precondiciones y uno de efectos. En los parámetros se indican los objetos que participan en la acción y su tipo, en las precondiciones se establecen las condiciones que se tienen que cumplir para el agente pueda realizar la acción, y en la parte de efectos aparecen las modificaciones que se realizaran sobre el estado una vez se ejecuta la acción. Como se puede ver la relación con la formalización proposicional es directa, la única diferencia es que no aparece el coste de la acción, esto es debido a que el coste de las acciones no se puede modificar sin utilizar alguno de los requerimientos más avanzados, es por ello que todas las acciones en PDDL por defecto tienen coste uno.

Para ver estas secciones de forma más clara a continuación se puede encontrar un fragmento de uno de los problemas de la IPC-2002, en este problema de tipo logístico se tienen que transportar varios paquetes en diferentes medios de transporte. Para que la explicación sea más clara el fragmento esta ligeramente modificado, cabe mencionar que los comentarios en PDDL vienen marcados por el símbolo ; .

```
;(I)
(define (domain logistics-strips)                ; Nombre del dominio que se representa
  (:requirements :adl :negative-preconditions)  ; Requerimientos utilizados
  ;(II)
  (:types city place physobj - object     ; City, place y physobj son object
          package vehicle    - physobj    ; Package, vehicle son physobj
          truck airplane     - vehicle    ; Truck y airplane heredan de vehicle
          airport location   - place      ; Airport y location heredan de place
  )
  ;(III)
  (:predicates
    (in-city ?loc - place ?city - city)   ; recibe como argumentos una ciudad y un lugar
    (at ?obj - physobj ?loc - place)      ; recibe como argumentos un physobj y un lugar
    (in ?pkg - package ?veh - vehicle)    ; recibe como argumentos un paquete y un vehículo
  )
  ;(IV)
  (:action load-truck
    :parameters (?pkg - package ?truck - truck ?loc - place)  ; Variables en la acción
    :precondition (and (at ?truck ?loc) (at ?pkg ?loc))       ; Predicados que se cumplen
    :effect (and (not (at ?pkg ?loc)) (in ?pkg ?truck))       ; Predicados que se modifican
  )
)
```

- (I) Asignamos logictic_problem como nombre de dominio y utilizamos los requerimientos negative-preconditions y adl. Negative preconditions permite utilizar el operador lógico not en las condiciones, tanto en los objetivos del problema como en las precondiciones de las acciones. En cambio el requerimiento ADL permite utilizar toda la sintaxis del lenguaje creado a finales de los 90 [28], para ello incluye de manera indirecta los siguientes requerimientos: Typing, Disjunctive Preconditions, Equality, Conditional Effects, Existential preconditions y Universal Preconditions .

- (II) Se crean los tipos de objetos que utilizaremos en el problema, la sintaxis para expresar la jerarquía es $< tipo\_heredero_1 >< tipo\_heredero_2 > ... < tipo\_heredero_n > - < Tipo\_del\_que\_heredan >$. En este caso city, place y physobj heredan del tipo object. Representando esta relación de forma gráfica obtenemos:

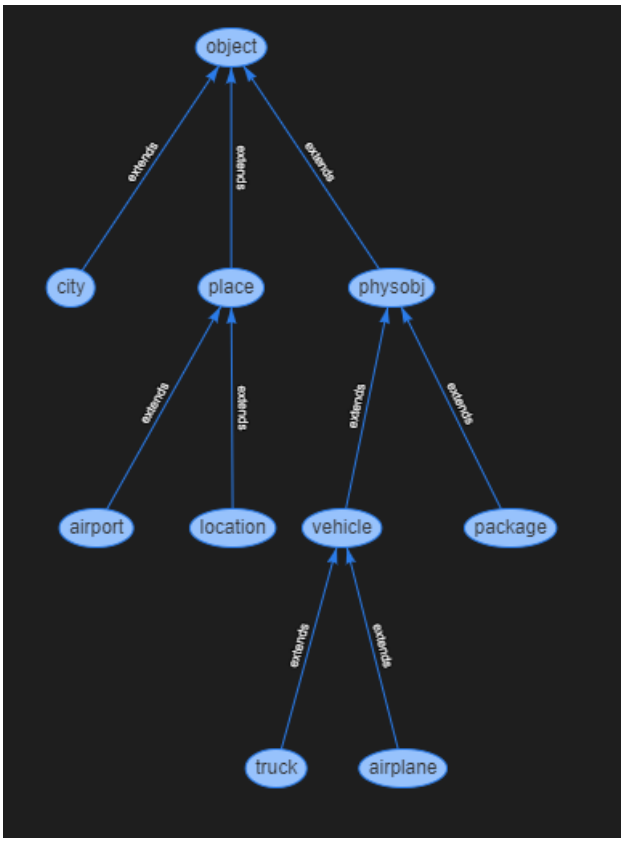

Figura 4: Representación de las herencias

- (III) Se definen los predicados, asignando a cada predicado un nombre y una serie de argumentos. Estos argumentos tienen la forma ?nombre_de_variable - tipo de argumento. Por ejemplo el predicado “atïndicará la relación entre un paquete y un vehículo, se utilizará para indicar si un paquete se encuentra en un vehículo.

- (IV) Cada una de las acciones tiene la siguiente estructura :action acompañado del nombre de la acción, :parameters expresando los parámetros igual que en los argumentos de los predicados, :preconditions donde se expresa una condición lógica mediante el uso de los predicados y :effect donde se indica el valor de los predicados que se modifican. En el caso de load-truck se necesita la existencia de un paquete, un camión y una lugar, tanto el paquete como el camión se tienen que encontrar en el lugar y el efecto es que el paquete ya no esta en la localización y se encuentra dentro del camión

Por otro lado en el fichero problema tiene la siguiente estructura:

- Meta-información del problema: A diferencia de en el fichero dominio solo es necesario definir el nombre del problema y especificar el nombre del dominio al que pertenece.
- Objetos: En este apartado aparecerán todos los objetos presentes en el mundo, debido a las restricciones de la planificación clásica y del lenguaje no se pueden crear ni eliminar objetos, así que estos serán los únicos presentes durante el desarrollo del plan. Para indicar el tipo al que pertenece cada objetos la sintaxis que se utilizara será $< objeto_1 > ... < objeto_n > - < tipo\_de\_objeto >$
- Init: En esta sección se define la situación inicial del problema, para ello habrá que indicar los predicados que se cumplen al comenzar el problema.
- Goal: En esta sección se indican los predicados que han de cumplirse en estado final del problema.

```
; Sección con metainformación:
(define
    (problem two_packages_problem)        ; Nombre del problema
    (:domain logistic_problem)            ; Dominio al que pertenece


    ; Conjunto de objetos:
    (:objects
            plane         - airplane      ; Existe un objeto de tipo airplane
            truck         - truck         ; Existe un objeto de tipo truck
            cdg lhr       - airport       ; Existen dos objetos tipo airport
            store1 store2 - location      ; Existen dos objetos tipo location
            paris london  - city          ; Existen dos objetos tipo city
            p1 p2         - package       ; Existen dos objetos tipo package
    )

    ; Estado inicial
    (:init (in-city cdg paris)            ; El aeropuerto cdg se encuentra en paris
           (in-city lhr london)           ; El aeropuerto lhr se encuentra en london
           (in-city store1 paris)         ; La localización store1 está en en paris
           (in-city store2 paris)         ; La localización store2 está en paris
           (at plane lhr)                 ; El avión está en el aeropuerto lhr
           (at truck cdg)                 ; El camión está en el aeropuerto lhr
           (at p1 lhr)                    ; El paquete p1 está en el aeropuerto lhr
           (at p2 lhr)                    ; El paquete p2 está en el aeropuerto lhr
    )

    (:goal (and (at p1 store1)            ; El objetivo es que el paquete p1 se
                (at p2 store2)            ; encuentre en la store1 y el paquete p2
           )                              ; en la store2
    )
)
```

2.3.2. Versiones oficiales de PDDL

Desde la creación de PDDL en 1998 han sido publicadas nuevas versiones del lenguaje, mejorando le expresividad del lenguaje y su flexibilidad. Esto ha permitido poco a poco representar problemas cada vez más alejados de algunas restricciones de la planificación clásica. Como hemos visto en el apartado anterior, estas nuevas versiones se han traducido en un aumento del número de requerimientos que se pueden utilizar para expresar un problema.

- PDDL 1.2: Nació en 1998, fue la primera versión de PDDL e incluye muchas de las expresiones que proponía ADL respecto a STRIPS. Además incluye sintaxis bastante avanzada como son los axiomas de dominio, los axiomas de dominio permiten definir predicados que implican otros predicados. Fue la versión utilizada durante la International Planning Competition de 1998 y la International Planning Competition del 2000.

- PDDL 2.1: Es una de las extensiones más importantes del lenguaje, nació en el año 2000 y permite realizar planificación concurrente y planificación de tiempo explicito, además se pueden tomar consideraciones numéricas a la hora de realizar el plan. Para tener en cuenta las consideraciones numéricas es necesario utilizar el requerimiento numeric-fluentes, este permite crear predicados numéricos que se pueden utilizar para crear condiciones y optimizar el plan desarrollado por el solver en función de estas variables. Por otro lado para realizar planificación de tiempo explicito, es necesario incluir el requerimientos durative actions, este permite definir la duración de acciones, a su vez el requerimiento durative inequalities permite el uso de desigualdades para poder expresar esta duración. Para expresar la planificación concurrente en el tiempo es necesario el requerimiento de continuous effects, este permite expresar las condiciones y los predicados en distintos puntos temporales de la acción. Esta versión hizo su primera aparición en la AIPSde 2002.

- PDDL 2.2: Es la tercera versión y fue lanzada en 2004, vuelve a introducir el concepto de los axiomas de dominio presente en PDDL 1.2 pero en forma de predicados derivados, cumplen la misma función que los aximoas pero permiten una sintaxis más cómoda y más limpia. También introduce el concepto de los literales iniciados en el tiempo, permitiendo modificar el valor de predicados en un instante concreto del tiempo, su utilidad principal es representar eventos externos. Fue la versión utilizada durante la International Planning Competition de 2004.

- PDDL 3.0: Es la cuarta versión y fue lanzada en 2005, permite la creación de restricciones intermedias a lo largo del plan, además de introducir las preferencias. Las preferencias permiten modelar los objetivos como metas duras o metas blandas, las metas duras son proposiciones que tienen que ser ciertas en el estado final para considerar que el plan es valido, en cambio las metas blandas no tienen porque cumplirse en el estado final pero incrementa la calidad del plan, cuando se definen las metas blandas también se indica cual es coste de no cumplirlas, permitiendo así que el planificador tenga una métrica con la que comparar distintos planes y su calidad. Fue la versión utilizada durante la International Planning Competition de 2006.

- PDDL 3.1: Es la versión actual, fue creada en 2008 e introduce los objetos variables. Con esta extensión las variables dejan de ser únicamente numéricas y pueden ser objetos de todo tipo, provocando un cambio bastante significativo semánticamente, además soporta STRIPS funcional. Fue el lenguaje oficial en la IPC de 2008 y 2011.

Dentro de la familia de PDDL se han ido creando otros lenguajes para expresar otros problemas de planificación alejadas de la restricciones clásicas, y otros para llevar un acercamiento distinto a los problemas clásicos, evitando así creando sintaxis compleja en PDDL, lo que acabría con su robustez y sistema de versiones:

- PDDL+ : Es una extensión de PDDL2.1 desarrollada entre 2002 y 2006, se centra en crear un modelo que contemple sucesos que ocurren obligatoriamente cuando las condiciones establecidas ocurren, esto lo logra mediante el uso de eventos y procesos. Los procesos son transiciones continuas en el tiempo que modifican una variable numérica, en cambio los eventos son transiciones instantáneas, estos dos cambios son bastante desafiantes para muchos planificadores, por lo que muchos planificadores pierden la capacidad de soportar algunas características de versiones anteriores de PDDL, para poder hacerse cargo de estas dos modificaciones.

- NDDL : New Domain Definition Language es la implementación de la NASA de PDDL 2.1, sus principales diferencias es que usan valores y variables para representar actividades en vez de lógica de primer orden o proposicional, la segunda es que no hay concepto ni de acción ni de estados, en vez de estos conceptos hay intervalos y limitaciones durante esos intervalos. Siendo un lenguaje con un modelo más parecido a un esquema de satisfabilidad booleana que a un esquema proposicional. En 2006 surgirá variante de NDDL llamada APPL (Abstract Plan Preparation Language), que permitirá hacer uso de las acciones junto a los intervalos.

- OPT :Ontology with Polymorphic Types es una extensión de PDDL2.1 en la que se intenta crear una notación de propósito general para crear ontologías, tienen una sintaxis muchísimo más elaborada y un sistema de tipos mas complejo, llegando a permitir utilizar expresiones del calcula Lambda o incluso estructuras de datos.

- MAPL: Multi-Agent Planning Language es una extensión de PDDL2.1 centrada en planificación multiagente, hereda muchos conceptos de MA-STRIPS entre los que destaca el concepto de privacidad, la privacidad son acciones o predicados que son propios de un tipo de agente, permitiendo separar roles entre distintos agentes. También añade variables de estado no proposicionales, permitiendo que los predicados se encuentren en otros estados que no sean verdadero o falso como desconocido o cualquier otra cosa que. Introduce la noción de eventos de PDDL+ y permite crear acciones cuya duración se determinará durante la ejecución del plan, y acciones explicitas para sincronizar el plan entre agentes, es decir acciones que permiten la comunicación entre agentes. MAPL también se puede usar como una extensión de PDDL3.0, permitiendo usar la sintaxis proposicional clásica de PDDL con nuevos requerimientos. Fue el lenguaje oficial de la IPC probabilística de 2011.

- MA-PDDL: Es una extensión de PDDL3.1 creada en 2012, se construye encima de esta versión a través del uso de requerimientos por lo tanto es completamente compatible con PDDL3.1, incluye el concepto de privacidad de acciones y predicados de MAPL, además de extenderlo a las métricas y a las metas, también permite crear precondiciones de acciones que no sean condiciones de predicadas si no que también se pueden establecer condiciones con otras acciones, lo que permite crear acciones concurrentes entre agentes consiguiendo una representación más general y flexible de la planificación multiagente.

- PPDDL: Probabilistic PDDL es una extensión de PDDL2.1 centrada en la planificación estocástica, entre sus nuevas características esta los efectos probabilísticos

que permiten que el efectos de las acciones pueda fallar siguiendo o probabilidad discreta o una función de distribución probabilística, también incluye otras maneras de presentar objetivos como los reward fluents, los goal rewards y los goal-achieved fluentes. Fue el lenguaje oficial de las IPC probabilísticas de 2004 y 2006.

- RDDL: Relational Dynamic influence Diagram Language se centra en la planificación estocástica, esta basado en PPDDL1.0 y PDDL3.0 pero aparte de traer consigo nuevas características importantes como la observación parcial de sucesos, trae consigo una sintaxis completamente distinta al resto de versiones.

2.4. ROS

El sistema operativo robótico es un framework ideado para desarrollar software en el campo de la robótica, este conjunto de herramientas surgió en 2007 y con el paso de los años ha ido cogiendo popularidad hasta convertirse en un estándar de facto dentro de muchos campos de la industria y de la robótica.

ROS pone a disposición del usuario una serie de bibliotecas que permiten crear un sistema con múltiples procesos de forma cómoda y en múltiples lenguajes de programación, un sistema de comunicación entre los distintos procesos, una serie de herramientas que permiten visualizar los que ocurre en el sistema, abstracción del hardware y control de dispositivos de bajo nivel. Es por estas funciones semejantes a las de un sistema operativo, por lo que se llama sistema operativo robótico. Entre sus principales ventajas destacan:

- Múltiples dispositivos: ROS permite la comunicación de procesos entre distintas maquinas que estén conectadas a una misma red. Esto permite la comunicación entres robots, la monitorización de robots desde un dispositivo externo o externalizar cálculos costosos fuera del robot.

- Open Source: ROS es de código abierto y gratuito lo que produce un incentivo para utilizarlo, lo que consigue una mayor comunidad.

- Múltiples lenguajes de lenguajes de programación: ROS da soporte de manera oficial a los lenguajes como python, C++ y Common Lisp, aunque permite y anima a la comunidad la creación de bibliotecas para lenguajes que no estén soportados oficialmente. Actualmente existen un total de 15 lenguajes con un soporte experimental entre los más utilizados: Java, Ada, Rust, JavaScript y Matlab.

- Ligero: ROS esta diseñado para ser lo más ligero posible, pone a disposición del usuario distintos paquetes de instalación entre estos el más ligero suele oscilar entre los 200 y los 400 MB dependiendo de la versión de ROS.

- Herramientas de desarrollo: ROS contiene numerosas herramientas que permiten un desarrollo de software más cómodo. Destacacan ROSTEST, un conjunto de herramientas enfocadas en los test unitarios y en los test de sistema, y los ROSBAG que permiten grabar la comunicación entre los distintos procesos del sistema y reproducirlos con posterioridad. Estos últimos son especialmente útiles para grabarlos mensajes que salen de los procesos encargados de los sensores.

- Multitud de paquetes creados por la comunidad: Al ser ROS de código libre parte de la filosofía que lo engloba provoca que multitud de usuarios suban las herramientas

que han desarrollado a repositorios públicos. Entre estos paquetes destacan simuladores como Gazebo y Rviz, drivers para todo tipo de sensores como cámaras kinect o sensores láser , herramientas de planificación y sistemas de navegación slam.

2.4.1. Historia de ROS

ROS es un proyecto desarrollado en 2007 por la incubadora William Garage y cuyos principales creadores son Keenan Wyrobek y Eric Berger, este sistema recoge muchas ideas de proyectos anteriores de robótica, entre los que destacan como principales antecesores el proyecto STAIR (STanford AI Robot) y el programa PR (Persoanel Robots). Estos dos proyectos iniciados a mediados de los 2000, tenían como objetivo desarrollar robots con cierto grado de autonomía e Inteligencia Artificial, durante su desarrollo ambos estudios crearon sus propios prototipos de sistemas de software dinámicos y flexibles destinados a la robótica. Tanto Keenan como Eric que participaban en estos proyectos, se dieron cuenta de los problemas que sufría la robótica en aquel entonces, estos problemas consistían en el elevado tiempo que requería reimplementar y rediseñar la infraestructura software necesaria para desarrollar algoritmos complejos en robots, como son los drivers de sensores y la comunicación entre distintos programas, lo que provocaba que hubiese muy poco tiempo para desarrollar sistemas inteligentes basados en esa infraestructura. Estas ideas serian recogidas por la incubadora estadounidense y se mejorarían hasta convertirse en 2009 lo que sería la primera versión de ROS1 llamada Mango Tango.

ROS no se quedo ahí, con el paso del tiempo se fueron creando versiones más avanzadas del framework, hasta conseguir que cada año saliera una nueva versión compatible con la versión anual de Ubuntu. Un gran cambio en la dirección de ROS se produjo en 2013 con la disolución de la incubadora William Garege, esto provocó que la Open Source Robotics Foundation se encargará de sostener el desarrollo de ROS. La Open Source Robotics Foundation es una corporación pública sin animo de beneficio creada para “Apoyar el desarrollo, distribución y adopción de software de código abierto para su uso en investigación, educación y desarrollo de productos en robótica ”. Este cambio en la dirección marcaría aun más el carácter del sistema, acercándolo aun más a las ideas colaborativas del OpenSource.

Actualmente ROS cuenta con un total de 14 distribuciones, estando ahora mismo soportadas ROS Melodic Morenia para Ubuntu 18.04 (Bionic) y ROS Noetic Ninjemys para Ubuntu 20.04 (Focal), además de contar con más de 86500 paquetes públicos creados por la comunidad. A pesar de ello Noetic será la última distribución de ROS1 dando paso a ROS2 un proyecto desarrollado por la Open Sourcce Robotics Foundation que surgio en 2015. ROS2 es una versión moderna de ROS que modifica bastantes partes del núcleo actual de ROS1, para poder incorporar muchos de los casos de uso que han aparecido estos años en el sector de la robótica. Actualmente desde la Open Source Robotis Foundation se soportan dos distribuciones de ROS2, Foxy Fiztroy diseñada para Ubuntu 20.04 (Focal) y Galactic Geochelone para Ubuntu 20.04 (Focal).

2.4.2. Arquitectura y conceptos

Un sistema basado en ROS esta formado por muchos procesos corriendo simultáneamente y comunicándose entre ellos mediante mensajes. La forma más clara de representar esta arquitectura es mediante un grafo, donde los nodos son los programas y los arcos son mensajes entre nodos. A continuación se muestra arquitectura ROS de un robot móvil el cual dispone de un brazo con dos grados de libertad y un sensor lidar:

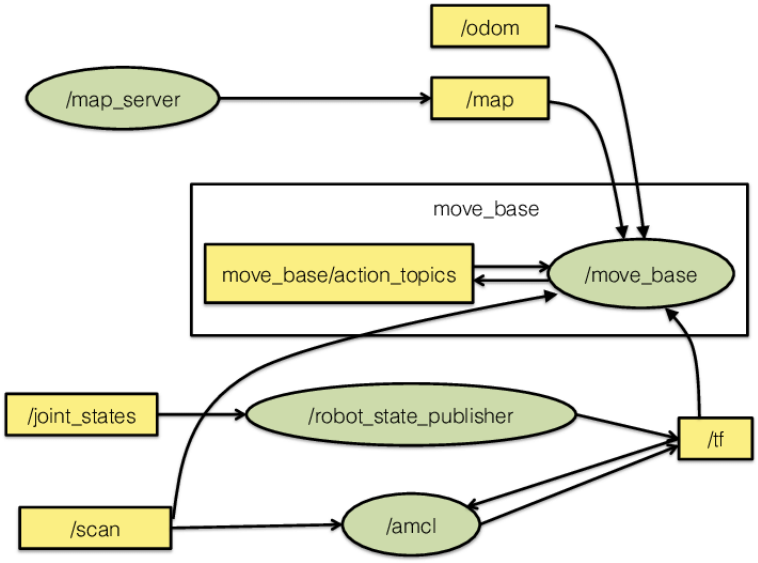

Figura 5: Arquitectura ROS de un robot móvil

Entre estos nodos hay un nodo especial llamado ROS Master, este nodo se encarga de gestionar la comunicación, cuando un nodo arranca se tiene que comunicar con él para indicar si espera algún tipo de comunicación, en caso de que tengan esa intención el ROS Master provee toda la información necesaria del resto de nodos que se encuentran activos, para que así cada uno de los nodos pueda establecer la comunicación punto a punto, además el ROS Master es el encargado de almacenar los parámetros del sistema.

Los topics permiten una comunicación asíncrona entre múltiples nodos, en este tipo de comunicación se distinguen dos roles el publicador y el suscriptor, el publicador es capaz de enviar mensajes a través del canal y el suscriptor recibe los mensajes, cuando un nodo se suscribe en un topic registra una función llamada callback, esta se ejecuta cuando llega un mensaje. Este tipo de comunicación es muy útil si se va a transmitir información de manera continua y no es una gran repercusión la perdida de algún mensaje, no es un comportamiento habitual la perdida de mensajes pero en este tipo de comunicación ROS no asegura que todos los mensajes sean recibidos. Un ejemplo donde este tipo de comunicación es muy popular es en el caso de sensores, es muy común que cada vez que se tiene un sensor crear un nodo que actué como driver, publique la información que va obteniendo del sensor mediante un topic, ya que la gran mayoría de los sensores en robótica consiguen información a mucha frecuencia y no se espera una variación muy grande entre medidas muy seguidas.

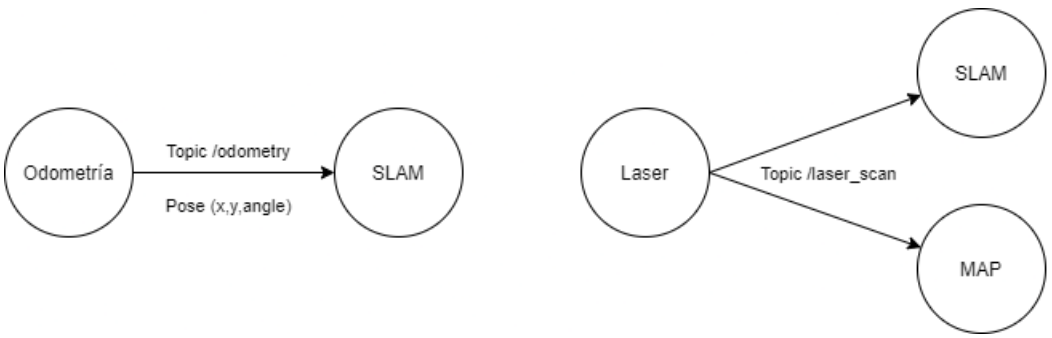

Figura 6: Funcionamiento de los topics

Los Servicios permiten una comunicación síncrona entre dos nodos, en el libro Pro-

gramming Robots with ROS [20] los autores lo definen como “llamadas a procesos remotas, que permiten a un nodo llamar a una función que se ejecuta en otro nodo”. En este tipo de comunicación se vuelvan a distinguir dos roles el servidor que provee el servicio y el cliente accede al servicio mediante un proxy local. En el caso del servidor es necesario definir una función de callback , esta será la función que se ejecute cuando el cliente haga la petición, cuando esta función se este ejecutando el nodo cliente se mantendrá a la espera de que el servidor responda, por ello es recomendable que la función callback sea una función limitada en el tiempo y lo más breve posible. Este tipo de comunicación se utilizan habitualmente para crear interfaces en algunos nodos o realizar acciones discretas con el robot, como por ejemplo activar un sensor dentro del robot o tomar fotos de alta resolución.

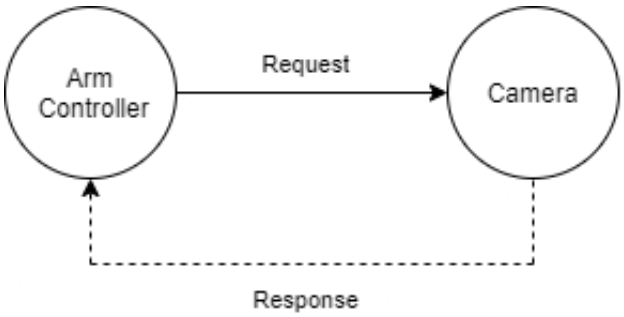

Figura 7: Funcionamiento de los servicios

Las acciones permiten una comunicación asíncrona entre dos nodos, son muy parecidas a los servicios, pero a diferencia de ellos evitan que el nodo que hace la petición se quede paralizado hasta obtener la respuesta del servidor. Es por ello que las acciones se utilizan para crear rutinas de alto nivel o comportamientos prolongados en el tiempo, como pedir al sistema de navegación que se desplace hasta una determinada coordenada o que un brazo robot se mueva hasta una posición muy precisa. Además este tipo de comunicación provee la capacidad de enviar mensajes de realimentación al nodo que hizo la petición, pudiendo conocer en todo momento la situación en la que se encuentra el robot respecto al objetivo, por otro lado las acciones incluyen un mecanismo que permiten al nodo cliente cancelar objetivos, por lo que analizar la información de los mensajes de realimentación es muy útil. Las acciones en ROS1 no vienen implementadas en la biblioteca básica del framework, esto es debido a que este tipo de comunicación no era esperada por el equipo de ROS en su nacimiento, y fue la comunidad la que se dio cuenta de que la comunicación para esta clase de rutinas eran muy difíciles de implementar con los servicios y los topics.

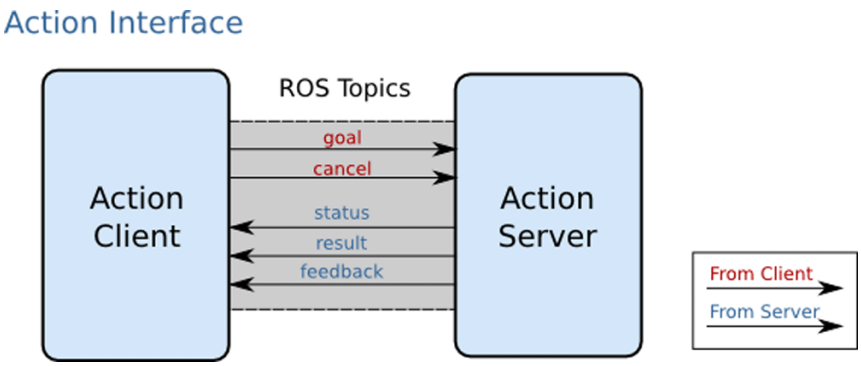

Figura 8: Funcionamiento de los servicios

ROS dispone de un servidor de parámetros al que los nodos pueden acceder para almacenar y recoger parámetros en tiempos de ejecución, este servidor esta conformado como

un diccionario multivariable y es el ROS Master el encargado de almacenarlo. Aunque este servidor no está diseñado para un alto rendimiento, es de gran utilidad para datos estáticos no binarios como son los parámetros de configuración. Este está destinado a ser visible globalmente de tal manera que las herramientas puedan inspeccionar fácilmente el estado de configuración del sistema, y modificarlo si es necesario. También es posible cargar datos directamente a este servidor desde un fichero yaml, por lo que es muy fácil de utilizar con otras herramientas de manera cómoda.

Los launchfiles o archivos de lanzamiento, son ficheros que permiten lanzar múltiples nodos a la vez, son muy útiles porque la forma estándar de levantar nodos en ROS solo te permite lanzarlos de uno en uno. Además los ficheros de lanzamiento permite establecer los parámetros del servidor de forma cómoda, e incluso permite modificar ciertos comportamientos de los nodos fácilmente, como si el nodo se tiene que levantar otra vez si deja de funcionar o por donde se tiene que enviar la salida estándar del programa.

2.4.3. ROS2

Con el paso del tiempo han cambiado muchísimas cosas en el sector de la robótica, y muchas de las ideas recogidas de los sistemas de mediados de los 2000, no pueden hacerse cargo de muchos de esos cambios ni muchos de los nuevos casos de uso que han ido surgiendo, por lo que durante los últimos años desde el equipo de desarrolladores de ROS se ha comenzado un nuevo proyecto llamado ROS2.

Entre sus principales ventajas encontramos:

- RCL: En ROS 1 para establecer la comunicación con la interfaz de ROS existe una biblioteca independiente para cada lenguaje de programación, estas bibliotecas se conocen como librerías cliente, esto provoca que muchos lenguajes les falten por implementar ciertas características de ROS o que directamente la forma en la que funcionan ciertos aspectos sea bastante distinto. ROS2 para acabar con esto ha creado RCL, una biblioteca en C que sirve de interfaz con ROS2 y que el resto de bibliotecas cliente se construyen sobre esta, permitiendo que todas las bibliotecas clientes puedan acceder a todas las ventajas de ROS2 y todas las innovaciones que vayan salido. Además de ser más sencillo el proceso para crear nuevas bibliotecas cliente, solo sería necesario crear las conexiones de cada lenguaje con RCL.

- ROS master: Ya no existe el concepto de ROS master, ahora los nodos son capaces de encontrar otros nodos sin su ayuda, permitiendo crear sistemas completamente distribuidos y evitando pasar por la centralización que provocaba. Esto a su vez genera que ya no exista un servidor de parámetros global, cada parámetro será especifico y único para cada nodo, pero se podrán modificar desde el exterior.

- Servicios y Acciones: Las acciones que en ROS1 no estaban incluidas en el núcleo de framework y trabajaban utilizando topics por debajo, en ROS2 tienen su propio protocolo de mensajería además de estar incluidos en el núcleo. Por otro lado los servicios en ROS2 pueden ser utilizados de forma asíncrona, lo que hace que el cliente no se encuentre bloqueado. Igualmente los servicios se sigue recomendando para acciones breves y las acciones para peticiones duraderas en el tiempo, esto es debido a que los servicios carecen de los mensajes de realimentación.

- LaunchFiles: En ROS1 los launchfiles se escribian en formato XML, lo que permitía un comportamiento muy poco dinámico a la hora de reconfigurarse. Ahora en ROS2

los launchfiles están escritos en python, por lo que permite una mayor flexibilidad y aplicar técnicas mucho más avanzadas.

- Soporte para más sistemas operativos: Las distribuciones de ROS1 estaban pensadas para sistemas linux tipo Ubuntu, aunque algunas distribuciones también eran compatibles con Debian e incluso Arch Linux. ROS2 soporta un mayor número de distribuciones linux, además de soportar otros sistemas operativos como Windows 10 y macOS 10.14.

3. Sistemas de planificación en ROS y en ROS2.

3.1. Rosplan

Rosplan es un paquete de ROS creado por el equipo de investigación de planificación computacional del King's College London, este paquete proporciona una serie de herramientas para implementar sistemas de planificación basados en la familia de lenguajes de PDDL, permitiendo realizar tanto la planificación, como la generación de problemas y la ejecución del plan. Además de permitir la replanificación en caliente durante la ejecución del plan y la validación de planes mediante el uso de la herramienta VAL.

Este paquete esta disponible para las versiones de ROS Kinetic Kame (Ubuntu 16.04) y ROS Melodic Morenia (Ubuntu 18.04), las funcionalidades de este framework están distribuidas en una serie de nodos con los cuales el usuario puede interactuar mediante el uso de diversos servicios de ROS. La forma en la que están conectados estos nodos se representa en el siguiente esquema, donde los óvalos representar los nodos de ROS y las cajas determinan las conexiones entre nodos mediante servicios, topics o parámetros.

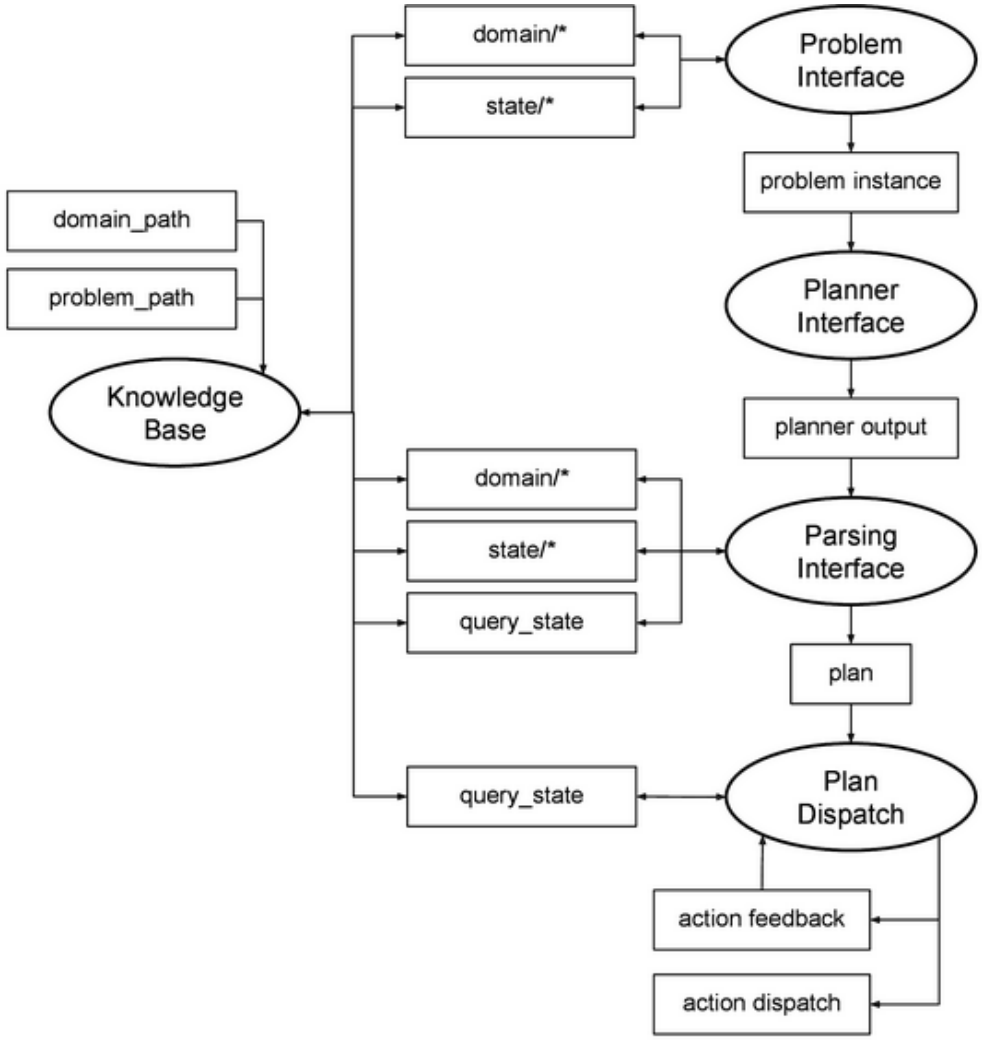

Figura 9: Arquitectura de Rosplan

A continuación se describe el funcionamiento de los distintos nodos:

3.1.1. Knowledge base

La base de conocimiento se encarga de almacenar el fichero del dominio PDDL y la instancia del problema, además de publicarla al resto del sistema mediante los topics domain_path y problem_path. Dispone de varios servicios que permiten modificar tanto la base conocimientos como la situación del problema, hacer consultas sobre el estado del problema e incluso permite eliminar el modelo PDDL almacenado por la base de conocimientos, permitiendo modificarlo en caliente. Entre los parámetros de configuración más importantes se encuentran el domain_path que hay configurar con la ruta al fichero del dominio y el problem_path con la ruta al fichero del problema.

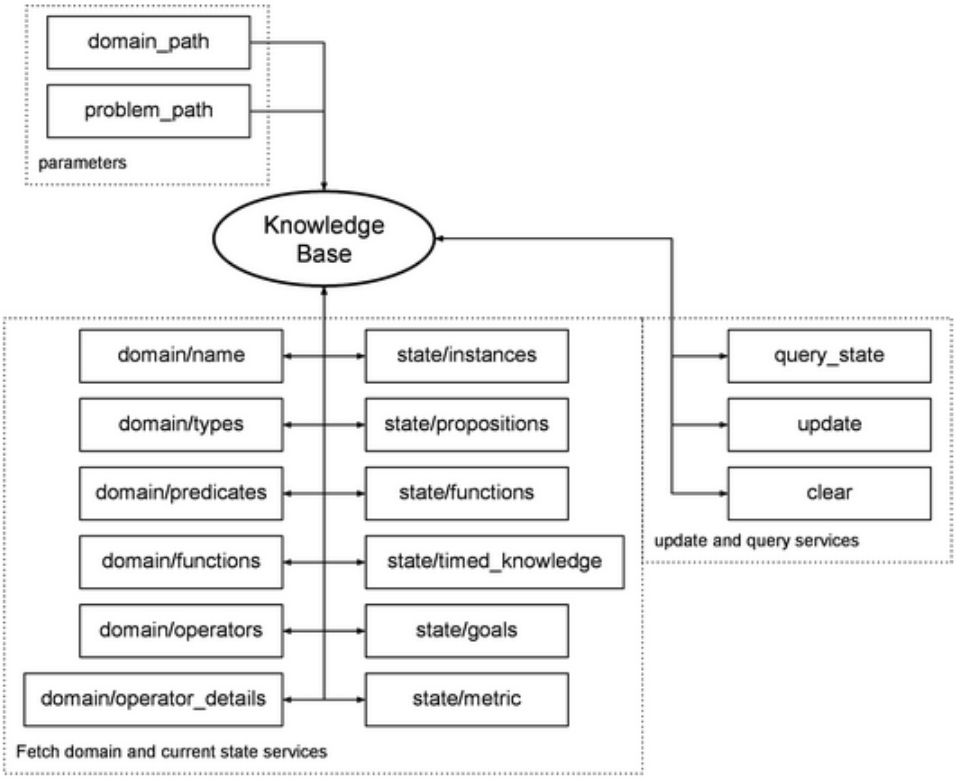

Figura 10: Diagrama de la base de conocimientos

3.1.2. Problem Interface

El nodo Problem Interface se encarga de realizar consultas sobre la instancia del problema a la base de conocimientos, publicar el estado actual del problema al resto de nodos en el topic problem_instance, y almacenar la instancia del problema en un fichero de texto externo, el cual será el que se entregue a los planificadores. Se almacena en un fichero externo para evitar modificar el archivo entregado por el usuario, ya que si se utilizase el original, al modificar la situación del problema se sobrescribirá. Puesto que el proceso que ejecuta este nodo es muy costoso y es poco demando, para activar el proceso de consulta existen dos servicios que actúan como interfaz:

- problem_generation_server: Inicia el proceso de consulta, almacena el problema en el fichero, y lo pública en el topic sin ninguna clase de modificación.

- problem_generation_server_params: Inicia el proceso de consulta a la base de conocimientos pero antes de almacenarlo y publicarlo, modifica los parámetros que ha enviado el usuario al hacer la petición de servicio. Permitiendo modificar el problema sin modificar la base de conocimientos.

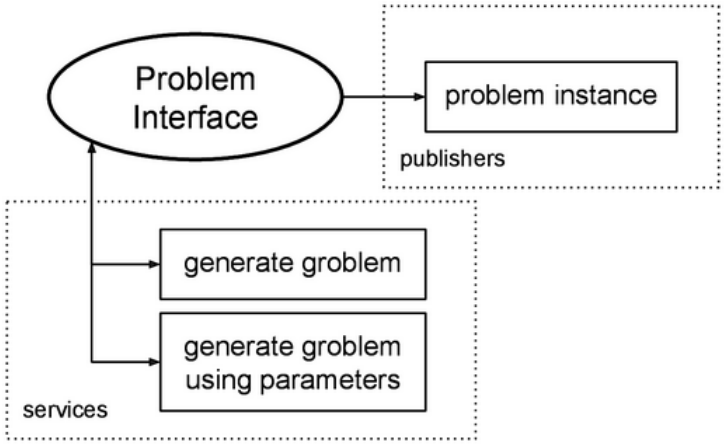

Figura 11: Diagrama del nodo Problem Interface

3.1.3. Planner Interface

El nodo Planner Interface envuelve al planificador, es el encargado de ejecutar el planificador cuando se hace una llamada por el servicio /planning_server, almacenar la solución del planificador en un directorio y enviar la la solución del problema a través de un topic. Es importante saber que cada planificador entrega la solución de maneras distintas, es por ello que no todos los planificadores se pueden utilizar en Rosplan ya que cada uno necesita de un nodo planner-interface distinto. Los planificadores de PDDL para los que existe interfaz son los siguientes: POPF, OPTIC, FF, Metric-FF, Contingent-FF, LPG, TFD y SMT. En el caso de RDDL solo existen interfaz para el planificador PROST.

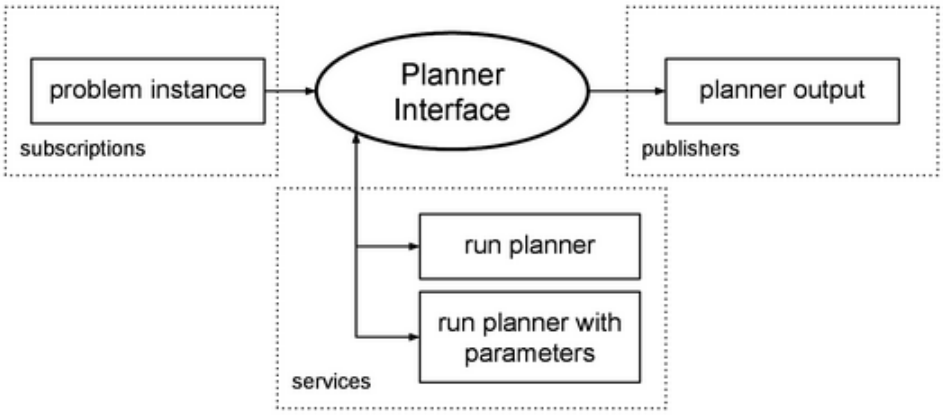

Figura 12: Diagrama del nodo Planner Interface

3.1.4. Parsing Interface

El nodo Parsing Interface convierte el plan generado por el planificador, que se encuentra almacenado como una cadena de caracteres, en una representación del plan que se puede ejecutar y cuyas acciones se pueden enviar a otras partes del sistema. Existen implementaciones de este nodo para crear representar planes como redes de Petri, planes esterel o planes secuenciales. Estas tres representaciones son bastante útiles para describir el plan generado por el planificador, pero dentro de ROS las mas utilizadas son los planes secuenciales y los planes esterel.

- Un plan secuencial se codifica en ROS como un vector de mensajes de acción, este vector esta ordenado para que los mensajes aparezcan en el orden del plan. Esta representación es suficiente para que más tarde el nodo de Dispatch sea capaz de dividirlo en distintas acciones si el plan es secuencial o temporal. Para planes con ejecución concurrente, el ejecutor requerirá un razonamiento adicional para determinar cuándo deben despacharse las acciones, por lo que solo se recomiendan esta representación para planes secuenciales.
- El EsterelPlan se codifica en ROS como un grafo compuesto por mensajes de acción. Los nodo pueden representar un inicio de plan, un inicio de acción o un final de acción, además tener un estado interno que puede representar que el nodo está en espera, en ejecución o finalizado. Un nodo de inicio de acción finaliza una vez que se envía la acción, mientras que un nodo final de acción se completa cuando se ha completado la ejecución de la acción. Las aristas representan ordenamientos causales o temporales en el plan, un nodo de inicio de acción solo se puede procesar una vez que se hayan completado todos los nodos ordenados anteriormente. Este tipo de mensaje se recomienda cuando existen acciones concurrentes.

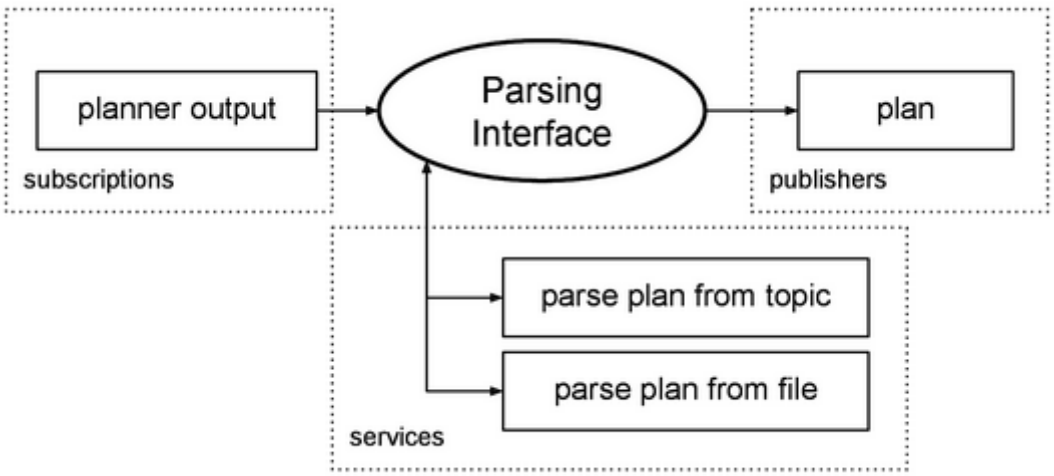

Figura 13: Diagrama de la interfaz del analizador sintáctico

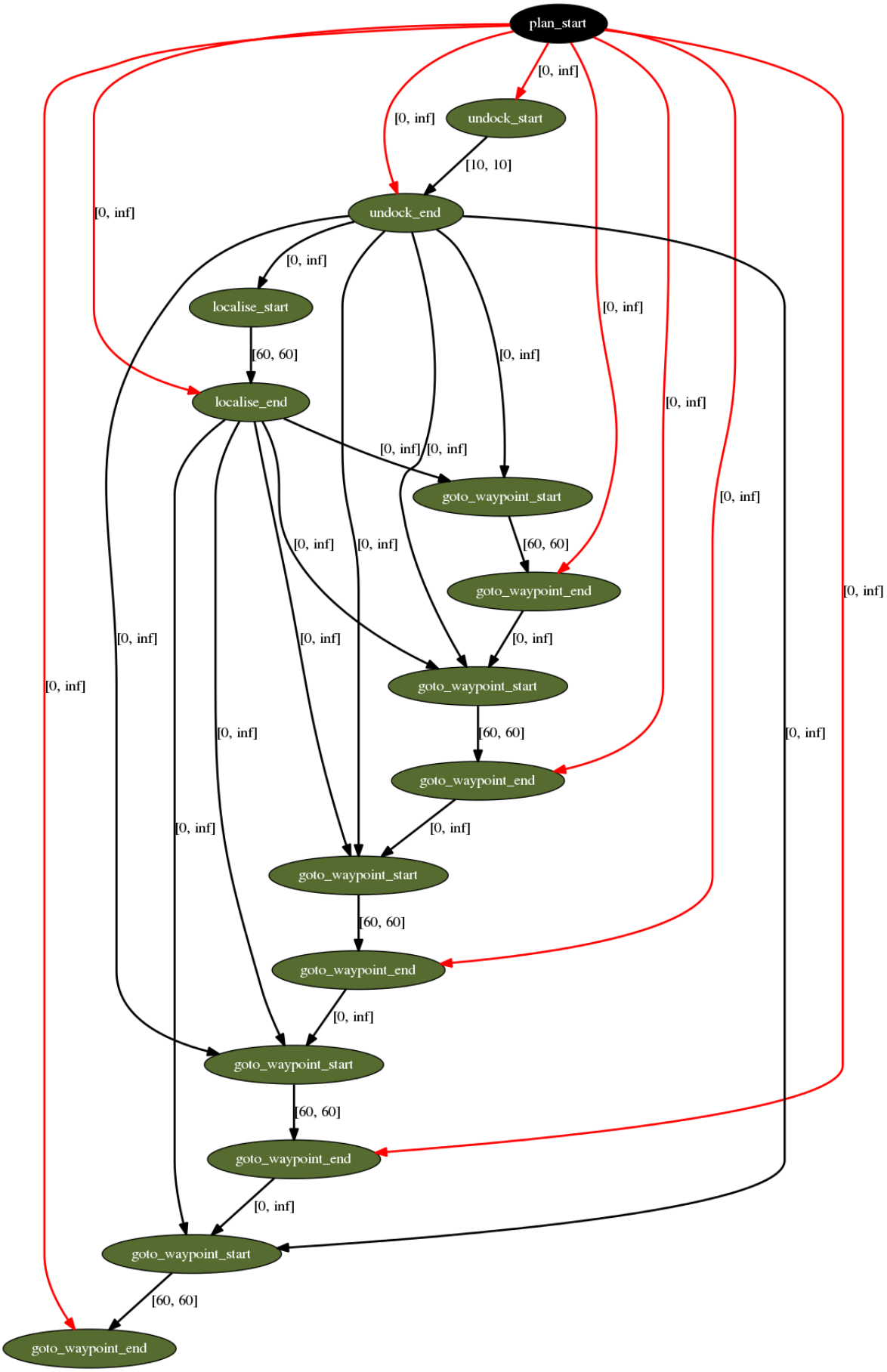

Figura 14: Representación mediante un plan ESTEREL

3.1.5. Plan dispatch

El nodo de Plan Dispatch se encarga de recoger la representación del plan y administrar su ejecución del plan. Para ello se comunica a través de dos topics con los nodos encargados de realizar las acciones, el topic /action_dispatch donde publica la acción que toca en cada momento, y el topic /action_feedback donde los nodos encargados de ejecutar la acción publican como se va desarrollando el plan. Internamente el funcionamiento de este nodo es altamente dependiente de la representación escogida en el nodo parsing interface.

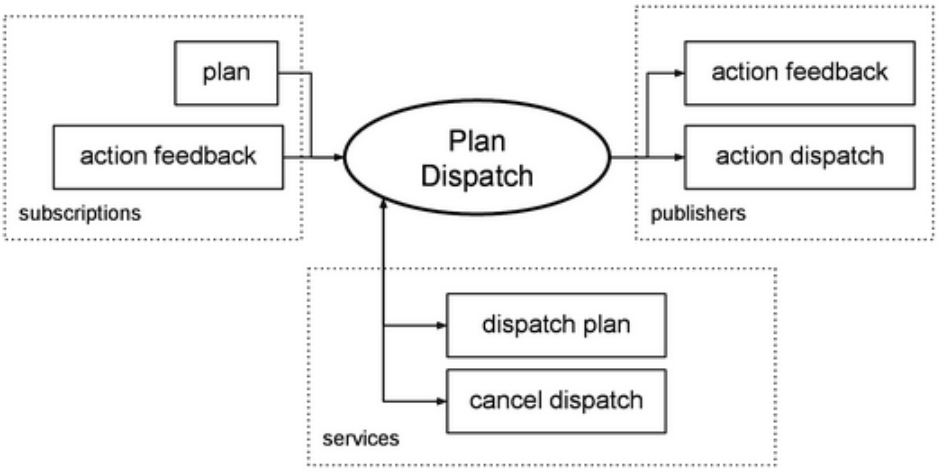

Figura 15: Diagrama del nodo Plan Dispatch

3.1.6. Action Interface

Como se ha podido observar en el plan dispatch es necesario que las rutinas del robot sean capaces de comunicarse con ROSPLAN, más concretamente con los topics del Plan Dispatch, para que este pueda gestionar la ejecución del plan, y con los servicios de la base de conocimientos, para actualizar el estado del problema a medida que se ejecuta. Todos estos canales de comunicación utilizan tipos de mensaje creados por Rosplan y codificados en ROS, siendo mensajes poco típicos en ROS y a veces poco cómodos de manejar. Es por ello que Rosplan pone a disposición del usuario una interfaz hecha en C++, que se encarga de la manipulación de todos estos canales de comunicación, y evitar así que el usuarios los tenga que gestionar manualmente.

3.1.7. Sensing Interface

Cuando se utilizan sistemas de planificación en entornos robotizados reales, es muy posible que las acciones no siempre tengan éxito y el estado del mundo puede cambiar por razones no relacionadas con el robot. Como sabemos los problemas de planificación clásicos no son capaces de representarlo ya que están muchos más restringidos. Y aunque haya lenguajes de acción que permitan expresar de manera más flexible, el lenguaje siguen sin poder expresar la complejidad de una situación real. Es por ello que cuando se utiliza Rosplan es necesario actualizar constantemente la base de conocimientos. Esta actualización la podemos hacer de dos maneras:

- Manera activa: Consiste en aprovechas el mecanismo de las Action Interface para actualizar la base de conocimientos. Para ello se pueden crear acciones en el dominio que representen el análisis de los sensores o fusionar alguna acción ya existente con el análisis de los sensores. Esta opción es útil para los sensores que no publican información habitualmente y llevan un alto procesamiento, como son las cámaras.

- Manera pasiva: Creando un nodo que se encargue de analizar los topics de sensores y actualice la base de conocimientos mediante los servicios disponibles. Este sistema es útil para aquellos sensores que publican información de manera periódica, como sensores láser o la odometría mecánica.

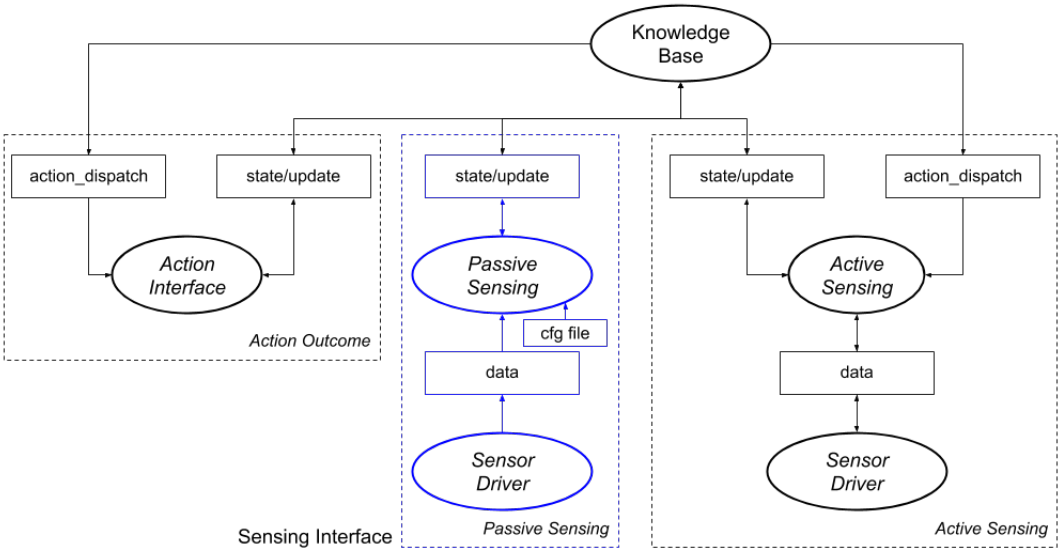

Figura 16: Diagrama de la interfaz del analizador sintáctico

Puesto que los mensajes de Rosplan son enfarragosos de manejar y el uso de servicios esta planteado para mensajes asíncronos, Rosplan vuelve a poner a disposición del usuario una interfaz para realizar estas acciones de manera automática y cómoda, la interfaz sensorial. A diferencia de la interfaz de acción que se implementa utilizando una clase de C++, la interfaz sensorial es un poco más compleja de configurar, siendo los siguientes archivos necesarios de para poder configurarla:

- Un archivo de configuración escrito en YAML que indicará de que manera se va a recibir la información de los sensores, pudiendo seleccionar entre suscribirse a un topic y/o hacer una petición a un servicio, como se va a procesar la información y que parámetros o predicados se van a modificar en la base de conocimientos. En el caso de que la información se obtenga haciendo petición a un servicio también se puede indicar la frecuencia a la que se realiza la llamada.

- Un script de python donde almacene funciones que se puedan utilizar para procesar información de manera compleja.

- El launchfile de ROS con el que cargar las configuraciones y el nodo encargado de leerla rosplan_sensing_interface

3.2. PlanSys2

Plansys2 es un paquete de ROS2 que permite integrar en la nueva versión del robótico sistemas de planificación basados en PDDL, es un paquete bastante reciente que saco su primera versión en 2019, e intenta aplicar los últimos conceptos del framework robótico.

Cuenta con bastantes de las funciones que tenía Rosplan y tiene una estructura similar, aunque pule algunas cosas como la excesiva cantidad de nodos que tenia el framework de ROS1 y la inmensa cantidad de mensajes, lo que lo hace más simple. Además a parte de tener un interfaz basada en ROS, trae consigo una librería cliente escrita en C++ que camufla el uso de los servicios, lo que permite que el usuario no tenga que ocuparse de la complejidad de este protocolo. El paquete esta distribuido en cuatro nodos el Domain Expert, el Problem Expert, el Planner y el Executor, encontrándose estructurado de la siguiente manera:

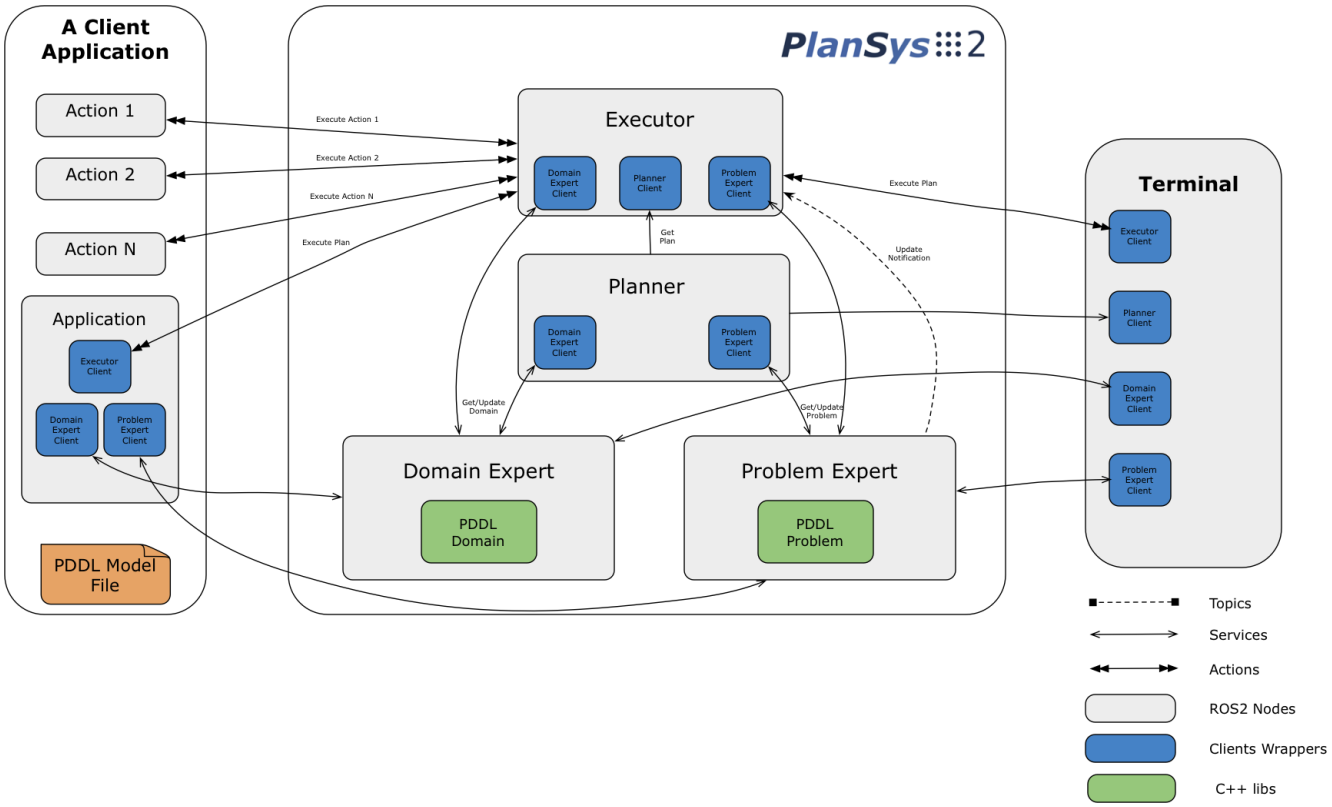

Figura 17: Arquitectura de Plansys2

3.2.1. Domain Expert

Este nodo se encarga de almacenar el contenido de los distintos ficheros dominio PDDL, además de permitir que el resto de nodos puedan acceder a él. En la interfaz de este nodo existen múltiples servicios para poder obtener información sobre el dominio, pero de momento no existen ninguno que permita su modificación, aunque los autores han anunciado que en un futuro existirá esta capacidad. Por otro lado este nodo se configura mediante un único parámetro el model_file , este recibe en formato string una o varias rutas a los ficheros de dominio que queremos que almacene, En el caso de entregar varios ficheros lo que hará el nodo será combinarlos en un único fichero de dominio. Esta posibilidad no existía en Rosplan, y permite que cada componente o sistema que utilice el robot tenga su propio dominio. Un ejemplo sería tener un problema de planificación exclusivamente

centrado en la navegación y otro centrado en el movimiento de los actuadores, cuando tuviésemos un robot que necesitase estos dos paquetes, no haría falta modificar los ficheros PDDL, ni levantar dos estructuras de Plansys2.

3.2.2. Problem Expert

Este nodo realiza las mismas funciones que el Domain Expert, pero con el fichero problema en vez de con el de dominio. Además incluye una interfaz para poder modificar el estado del problema, estos cambios los notifica al resto del sistema utilizando dos topics.

3.2.3. Planner

Es el encargado de ejecutar el planificador cuando recibe una petición por un servicio o por la interfaz de C++, antes de llamar al planificador este nodo hace una petición al Domain Expert y al Problem Expert para almacenar los correspondientes ficheros en el directorio /tmp, después llama el planificador y almacena el resultado en otro fichero en la carpeta /tmp para enviarlo más tarde. Este nodo tiene un parámetro configurable el plan_solver_plugins, sirve para indicar que planificador queremos utilizar para resolver el problema. Actualmente están disponibles el POPF solver y el Temporal Fast Downward, a la hora de configurar este parámetro tenemos que entregar una cadena con el nombre del pluggin que envuelve al planificador. Un pluggin de ROS es una clase de C++ que se puede acoplar y desacoplar dinámicamente en tiempo de ejecución, lo que permite modificar el comportamiento del nodo sin necesidad de cambiar el código fuente.

3.2.4. Executor

Este componente es el encargado de ejecutar el plan, para ello una vez recibido el plan desde el componente planner analiza el plan y administra su ejecución. Esta administración la realiza llamando a los distintos nodos responsables de cada acción, permitiendo que se puedan ejecutar varias acciones cuando es posible y permitiendo que una acción pueda ser realizada por más de un agente. Además este nodo realiza una confirmación de que las precondiciones se cumplen antes de realizar cada acción .

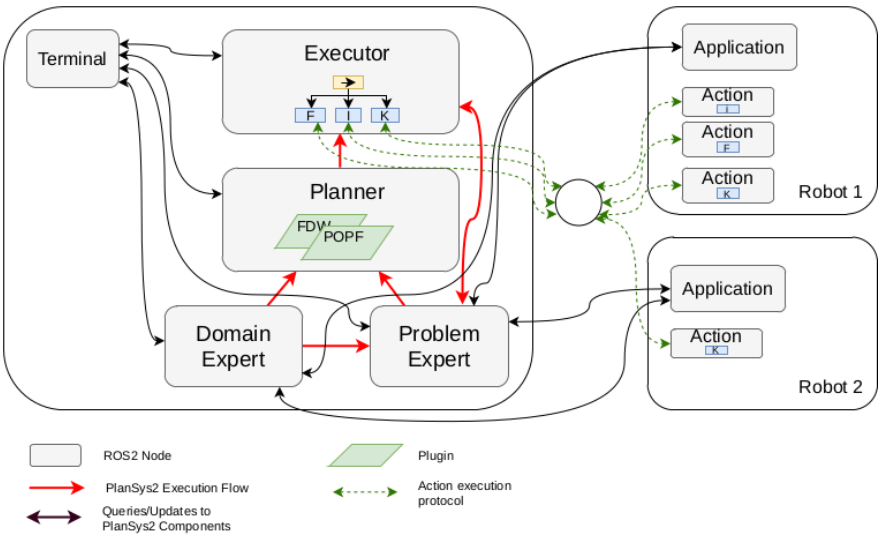

Figura 18: Plansys2 flujo de ejecución

Entre los parámetros de este componente destaca action_timeouts.actions, la activación de este componente permite configurar varias acciones para que se detengan si se retrasan por encima de un porcentaje máximo del valor previsto.

Estas optimizaciones las consigue construyendo un árbol de comportamiento una vez es recibido el plan. Para ello realiza los siguientes pasos:

1. Construir un grafo de planificación que codifique las dependencias entre acciones. Estas dependencias las halla a través de ver la relación entre los efectos de una acción y las precondiciones de las futuras acciones.

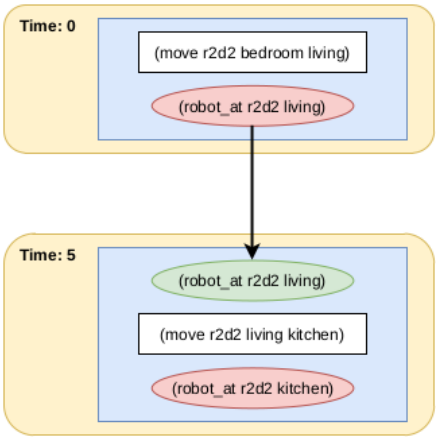

Figura 19: Fragmento del plan secuencial

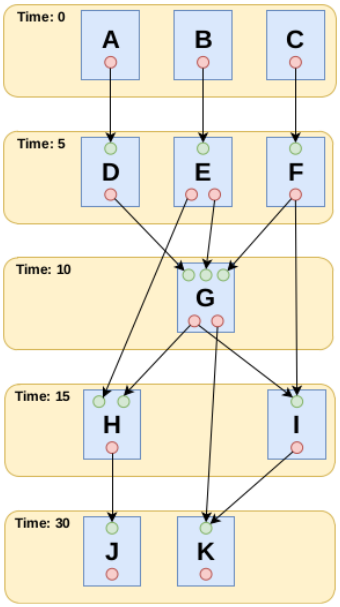

Figura 20: Grafo de planificación

2. Una vez creado el grafo identifica los flujos de ejecución y crea los arboles de comportamiento.

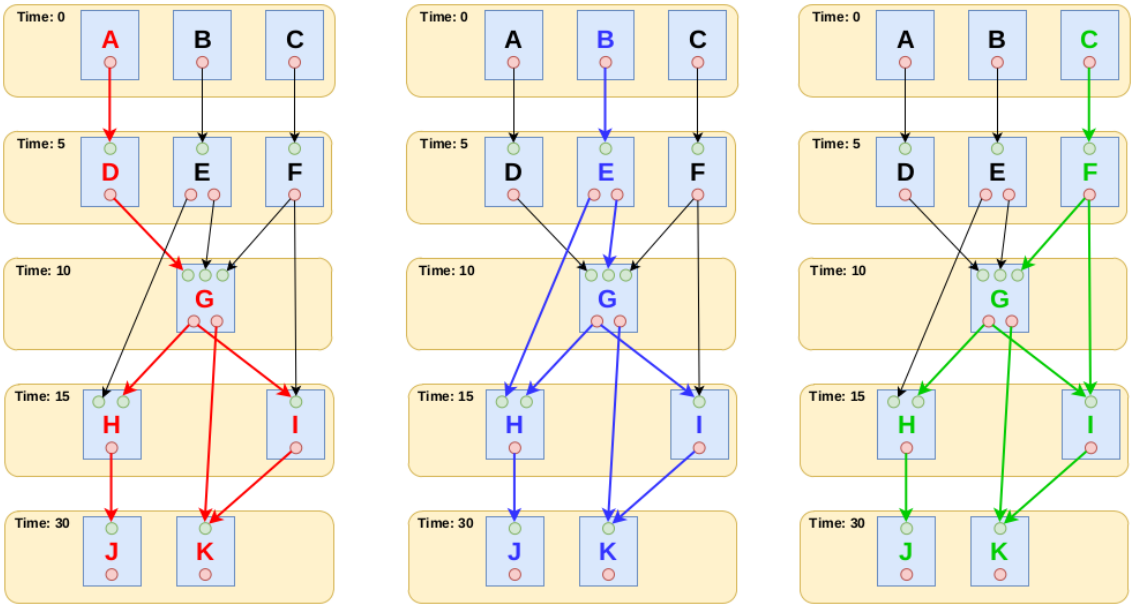

Figura 21: Identificación de los flujos de ejecución

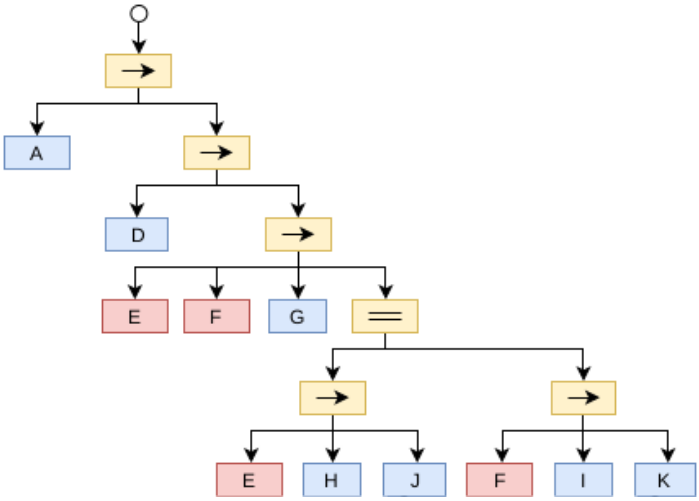

Figura 22: Árbol de comportamiento creado

3.2.5. Action delivery protocol

En la primera versión de Plansys2, cuando el componente Executor quería comunicarse con un nodo para que empezará a realizar una acción, este proceso empleaba el protocolo de Acciones de ROS2. Este protocolo ha sido descartado por los desarrolladores por no ser sufcientemente flexible, a cambio han desarrollado un protocolo basado en pujas, en este protocolo cuando el ejecutor tiene que realizar una acción, pregunta por el canal si algún agente puede ejecutarla, si varios agentes responden que pueden realizarla el ejecutor escoge al primero que respondió, si nadie responde el Executor realiza la petición hasta que se hace una petición para cancelar el plan.

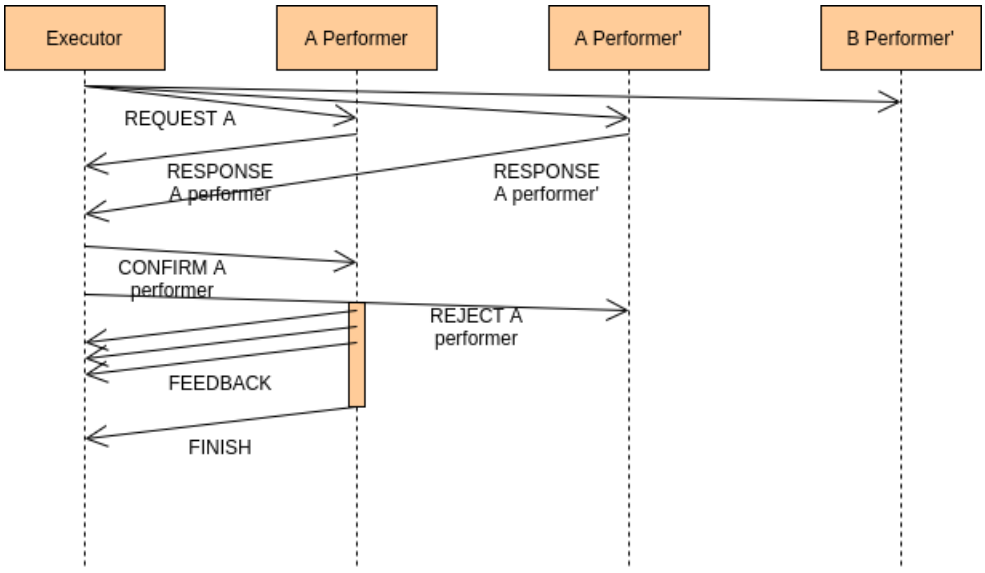

Figura 23: Plansys2 flujo de ejecución

Para poder configurar este protocolo en los nodos encargados de realizar las acciones del plan, existen una clase de C++ que hace de interfaz, este clase tiene dos parámetros el parámetro /action de tipo de string en el que se almacena la acción que es capaz de realizar el nodo, y el parámetro /specialized_arguments si este parámetro no es nulo, solo responde a la solicitud de acción que contiene en cualquiera de los argumentos alguno de estos valores.

4. Integración en entornos robotizados mediante Rosplan

Para comprobar la utilidad que presenta la planificación automática en entornos robotizados vamos a empezar integrando estas técnicas en un escenario bastante común de la robótica, la logística, nuestro primer escenario va a ser un almacén donde un robot, el agente, va a tener que recoger distintos cubos y llevarlos hasta distintos lugares, en este tipo de aplicaciones nuestra principal métrica sera recoger los paquetes de la forma más rápida posible. Este primer entorno se desarrollará utilizando ROS1 Melodic y Rosplan.

4.1. Entorno

Vamos a utilizar un escenario muy parecido al planteado en el libro Programming Robots with ROS [20], en este escenario vamos a disponer de varias salas con una serie de estanterías que se encuentren a la altura del brazo robot, sobre estas estanterías se van a encontrar los objetos que tiene que recoger el robot, o sera donde los tendrá que dejar. Cada una de las estanterías estará numeradas con un marcador que se encontrará en la parte superior, este marcador permitirá que el robot sea capaz de coger el objeto con mayor precisión y quitarle responsabilidad al sistema de navegación. Este escenario será recreado mediante el simulador Gazebo, lo que nos dará ciertas ventajas a la hora de recrear el robot como se comentará más adelante.

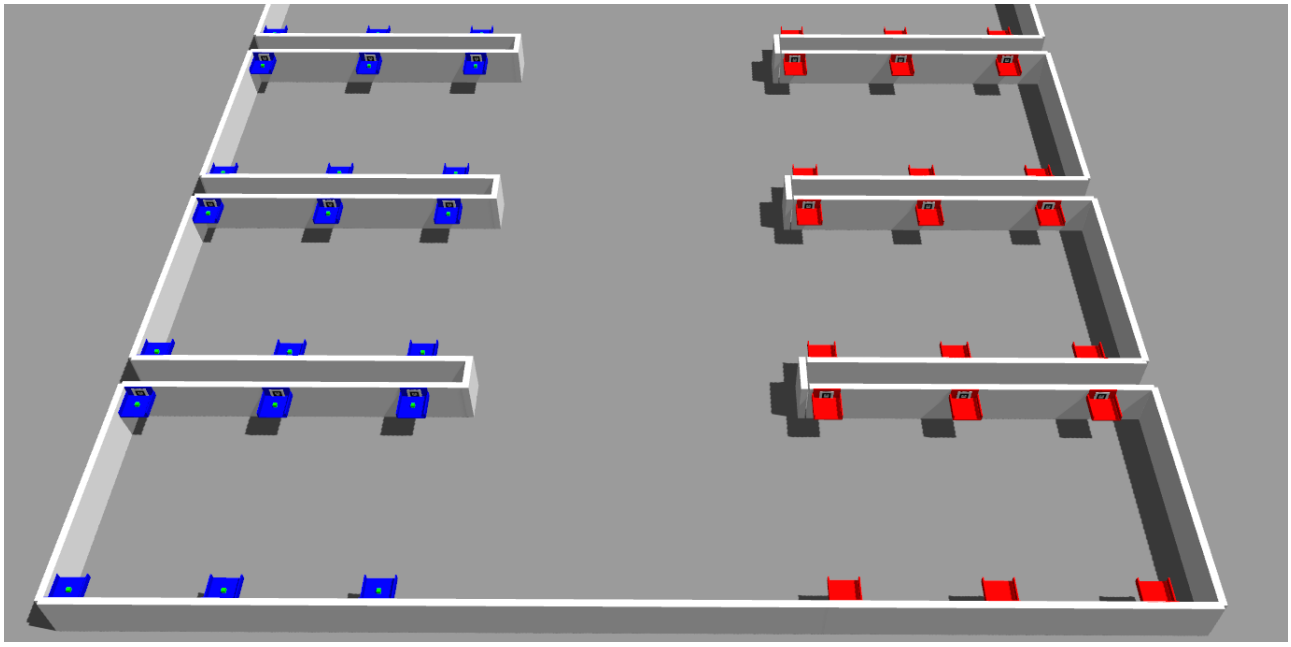

Figura 24: Almacén diseñado

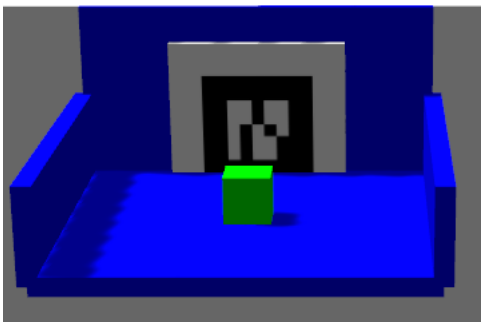

Figura 25: Estantería diseñada

En cuanto al robot, el robot que vamos a utilizar es el manipulador movil Fecth de Fetch Robotic, este robot es un heredero del PR2 de Williom Garage, fue diseñado para ser robusto tener un alto rendimiento y un bajo costo, se pueda utilizar para aplicaciones comerciales aunque esta diseñado para que ser utilizado principalmente en entornos de investigación y desarrollo. Este robot cuenta con las siguientes características:

- Sistema de tracción: Cuenta con dos motores sin escobillas en una configuración diferencial, teniendo cada motor un enconder en el eje, además la plataforma cuenta con 6 ruedas caster para mantener la estabilidad. Esta configuración le permite alcanzar el metro por segundo.
- Brazo poliarticulado : Cuenta con un brazo de 7 grados de libertad con una capacidad de 6Kg, con una extensión de 940.5mm y con una velocidad máxima de 1 m/s. Además cuenta con un gripper como efector final, este puede realizar una fuerza de 245N y cuenta con una apertura máxima de 100mm, más que suficiente para poder realizar las pruebas.
- Lidar SICK: Este robot cuenta con un sensor láser 2D con 25 metros de alcance situado en la parte intermedia del robot y con una capacidad para ver 220 grados sexagesimales. Esto lo convierte en una muy buena herramienta para evitar obstáculos y para la localización en casi cualquier entorno.
- Cámara Primesense Carmine : Es una cámara de profundidad con un rango corto de 1.09 metros y con una capacidad de 30 fotogramas por segundo, se encuentra en la cabeza del robot y esta calibrada para observar entre 0.35 metros y 1.4 metros. Todo esto le permite ser de gran ayuda en la navegación pero sobre todo permite al robot un conseguir una manipulación correcta.
- Unidad de medición inercial (IMU): Además de los sensores de largo alcance, el fetch cuenta con dos IMUs que constan tanto de un acelerómetro y un giroscopio de 3 ejes, un IMU se encuentra en la base del Fetch y se utiliza en combinación con la odometría en Unscented Kalman Filter (UKF) para mejorar la localización. El otro IMU se encuentra el gripper para conocer su estado con mayor precisión
- Ros: Toda la arquitectura software del robot, a excepción de algunos drivers y Firmware, esta construida encima de ROS lo que hace que disponga de todas las ventajas del framework y sea fácilmente manipulable .

Se ha escogido este robot porque además de cumplir con las características necesarias para la manipulación y el transporte de objetos, cuenta con un paquete de ROS que contiene un modelo de simulación para el simulador Gazebo, lo que hará más sencillo el desarrollo y la implementación.

Figura 26: Características del robot Fetch

4.2. Arquitectura software

La arquitectura presente en el robot es una adaptación de la arquitectura del agente inteligente presentada en la introducción. Este sistema esta dividido en tres capas o niveles, donde cada capa solo puede comunicarse con la capa superior o con la capa inferior, las tres capas están organizadas de manera que van desde un nivel de abstracción menor hasta uno mayor.

1. Drivers y controladores: Son los nodos encargados de comunicarse con los distintos sensores y actuadores.

2. Integración sensorial y planificadores dependiente del dominio: Es el encargado de ejecutar el plan entregado por el sistema de planificación e indicar a los actuadores como realizarlo, además ha de analizar la información entregado por los drivers de los sensores para conseguir la representación más veraz del entorno.

3. Sistema de planificación: Será el encargado de calcular cuales son los acciones que ha de realizar el robot para cumplir con su objetivo.

Como antes se ha mencionado al sistema de planificación se desarrollará mediante el paquete Rosplan, y su desarrollo e implementación se describirá de forma pausada en los siguientes apartados. En lo que respecta a la capa 1 y a la capa 2 utilizaremos varios de los paquetes creados por la comunidad de ROS, muchos de estos paquetes han sido desarrollados y son mantenidos por empresas dentro del ámbito industrial y la robótica, por lo que presentan grandes fortalezas. Como los paquetes presentes en estos dos niveles no son parte del ámbito central del estudio de este trabajo fin de grado, simplemente resumiremos por encima sus características pero no hablaremos del proceso de implementación y configuración que conlleva utilizarlos:

- Gazebo drivers: Gazebo es uno de los entornos de simulación más importantes de ROS, además de poder crear distintos escenarios con los que simular comportamientos de modelos físicos, gazebo permite crear drivers virtuales. Estos son pequeños componentes software que podemos asociar a nodos de ROS y a componentes físicos dentro del modelo del robot, permitiendo simulen tanto sensores como actuadores. Utilizaremos este paquete para los componentes en el nivel uno de nuestro arquitectura, siendo necesarios los siguientes driver para recrear la estructura del robot Fetch un driver para cada uno de las articulaciones del robot incluido el gripper y el sistema motor de la cabeza, un driver para el sistema de tracción diferencial, un driver para la cámara del robot, un driver para cada uno de los IMUS y uno para el sensor Lidar.

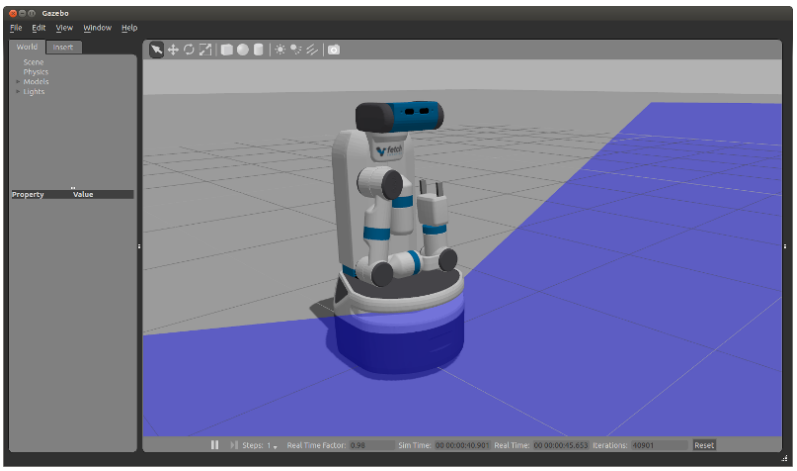

Figura 27: Entorno de simulación Gazebo

- MoveIt: Es un planificador de trayectorias para actuadores robóticos con numerosos grados de libertad, es uno de los paquetes más importantes dentro ROS y empezó siendo desarrollado por Williom Garage . Además se puede entregar una representación de un robot en formato URDF, formato utilizado para describir el modelo de un robot en ROS, y este paquete es capaz de generar el modelo cinemático y genera las arquitectura de control necesarios para ejecutar las trayectorias del planificador. Este paquete se utilizará dentro de nuestra arquitectura para controlar el brazo de forma cómoda, evitando tener que calcular manualmente la cinemática inversa y las estructuras de control.

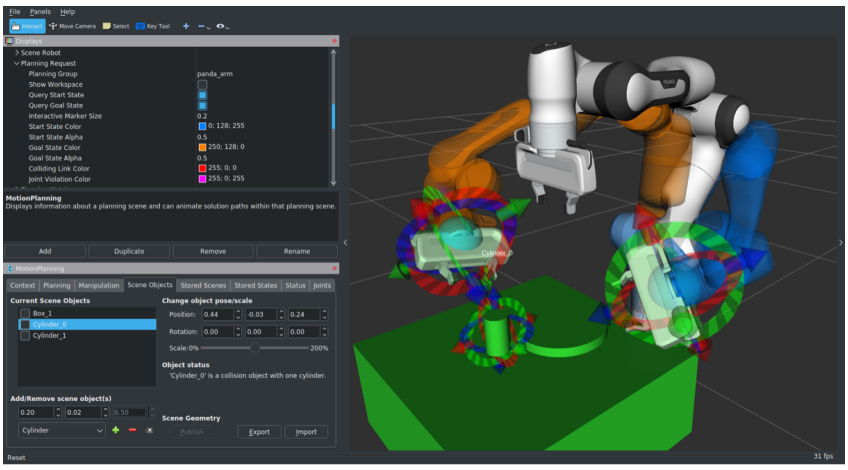

Figura 28: Análisis cinemático del paquete MoveIt

- Nav stack: El stack de navegación de ROS es uno de los paquetes más utilizados de todo el framework, este paquete es capaz de aplicar técnicas de integración sensorial, generar un control de velocidad sobre la planta de tracción y generar rutas de navegación. Este sistema de navegación tiene ciertas restricciones hardware como son tener una planta de tracción diferencial o holonómica, y tener a parte de la odometría algún sensor láser que permita la detección de obstáculos, además para el calculo de trayectorias es necesario disponer de un mapa del entorno. Este paquete será utilizado para la navegación del robot entre los distintos puntos de la sala.

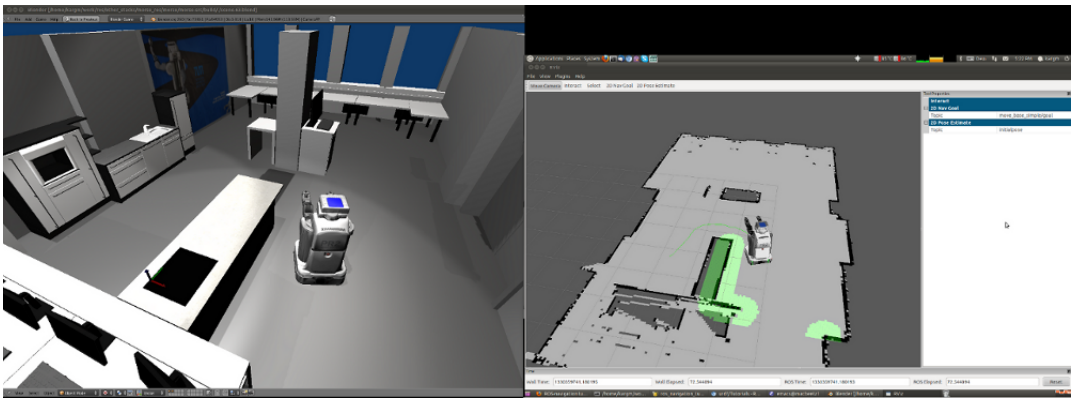

Figura 29: Sistema de navegación Nav Stack

- Map_server: Map server es una utilidad que permite crear mapas de ROS utilizando los sensores de los propios robots, trae consigo múltiples herramientas que permiten su visualización y su modificación forma dinámica o por el usuario posteriormente. La principal utilidad de este nodo sera crear el mapa de la instalación logística, actualizarlo dinámicamente cuando el robot este activado y publicarlo en un topic para que otras herramientas, como el navegation stack, tengan acceso a el mapa.

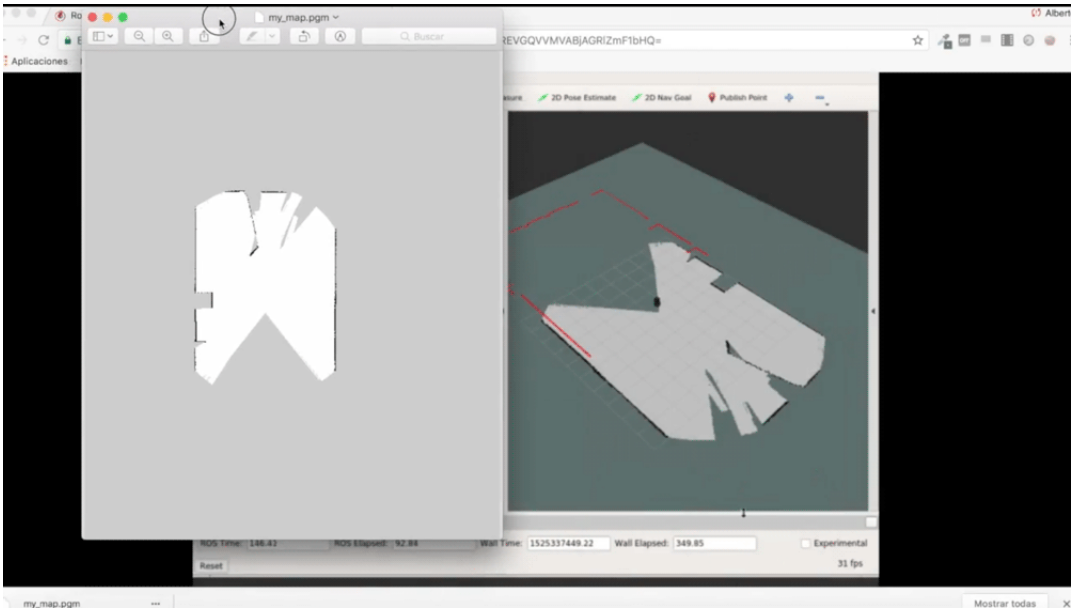

Figura 30: Soporte del mapa realizado por el Map server

- ar_track_alvar: Este paquete contiene herramientas para el reconocimiento de marcadores utilizando técnicas de visión artificial. Utilizaremos este paquete para poder identificar los códigos de las distintas estanterías y calcular la posición del robot frente a ella.

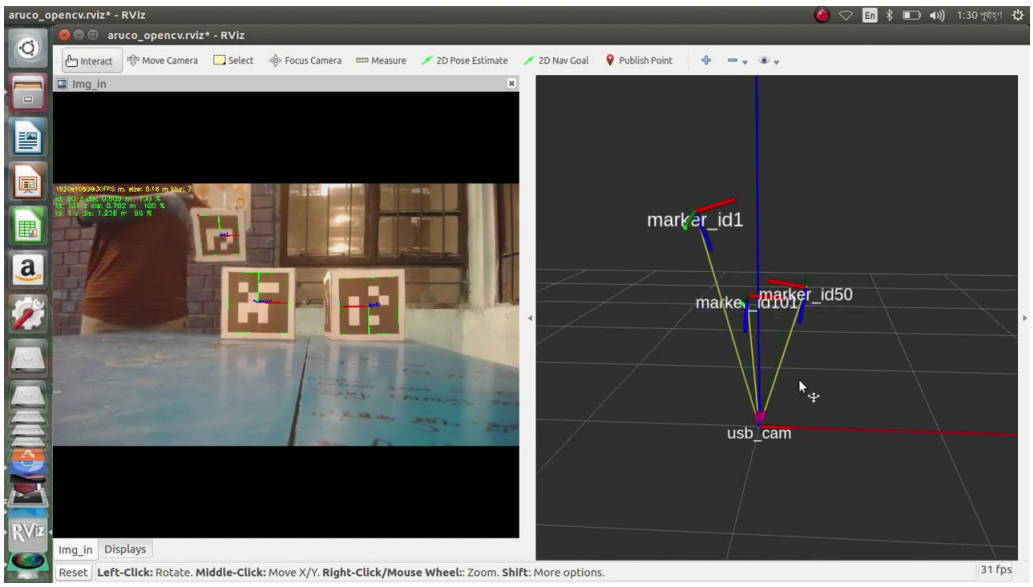

Figura 31: Detección de marcadores por el paquete ar_track_alvar

4.3. Representación del problema mediante PDDL

A la hora de demostrar las virtudes de PDDL, vamos a ir generando dominios y problemas que vayan pasando desde los comportamientos más primitivos, hasta comportamientos más avanzados que puedan llegar a sustituir el comportamiento en centros logístico. Empezaremos con una planificación que permita calcular los pasos para recoger distintos objetos y colocarlos, y después iremos incorporando otras herramientas más avanzadas de PDDL2.1 para ver cual es el plan más eficiente para recoger ciertos objetos y colocarlos en cierto orden.

Vamos a empezar utilizando PDDL1.2, la versión más básica de PDDL, lo primero que vamos hacer es definir cuales son los comportamientos de alto nivel que es capaz de realizar el Fetch en este entorno. Fácilmente podemos caracterizar tres acciones muy distintas moverse de un lugar a otro, coger una caja y portarla o dejar la caja que porta en un lugar. Sabiendo esto ya somos capaces de definir al menos tres acciones en el fichero de dominio de PDDL y podemos razonar sobre que tipos y que predicados habrá en nuestros ficheros. Las acciones llevan incorporadas tres tipos de objetos, el robot, las cajas y un lugar donde ir y venir, como tanto el robot como las cajas comparten la capacidad estar en un lugar, es correcto asumir que pueden compartir una herencia de una clase, también es fácil prever que necesitaremos un tipo lugar y un predicado que relacione el lugar donde están los objetos con los propios objetos, a ese predicado lo llamaremos “en”. Por otro lado crearemos otro predicado llamado “portada”para indicar si una caja esta siendo transportada por el robot. Como el robot tiene un limite de peso pequeño, solo puede cargar 6 Kg, por el momento solo permitiremos al robot llevar una única caja, por lo que crearemos un tercer predicado que indique que el robot este ocupado y no pueda recoger más cajas. Quedando de la sección de tipos y de predicados de la siguiente manera:

```
; Sección de tipos
(:types robot caja  - objeto
        lugar
)
```

```
; Sección de predicados:
(:predicates
        (en       ?obj - objeto  ?loc - lugar)
        (portada ?obj - caja    ?robot - robot)
        (ocupado  ?robot - robot)
)
```

Ahora que tenemos los predicados y los tipos las acciones se escriben de manera casi directa.

- Moverse: Incluye tres parámetros el robot que se mueve, el lugar de partida y el lugar de llegada. La precondición de esta acción es que el robot tienen que estar en el lugar de partida y el efecto es que el robot deja de estar en el lugar de partida y esta en el lugar de llegada. Codificándolo:

```
(:action moverse
    :parameters (?robot - robot ?from ?to - lugar)
    :precondition (and
        (en ?robot ?from)
        (not (en ?robot ?to))

    )
    :effect (and
        (en ?robot ?to)
        (not (en ?robot ?from))
    )
)
```

- Recoger caja: Incluye tres parámetros el robot, la caja y un lugar. La precondición es que tanto el robot como la caja tienen que estar en el mismo lugar y el robot no tiene que llevar ninguna otra caja, es decir no esta ocupado. El efecto es que la caja deja de encontrarse en el lugar donde estaba y es portada por el robot, estando el robot ocupado. Codificándolo:

```
(:action recoger_caja
    :parameters (?r - robot ?obj - caja ?loc -lugar)
    :precondition (and
        (en ?r ?loc)
        (en ?obj ?loc)
        (not (ocupado ?r)))
    :effect (and
        (not (en ?obj ?loc))
        (portada ?obj ?r)
        (ocupado ?r))
)
```

- Dejar caja: Incluye tres parámetros el robot, la caja y un lugar. La precondición es que el robot tiene que portar la caja y estar en el lugar donde quiere dejarla además de estar ocupado. Y el efecto es que la caja deja de ser portada por el robot, quedando el robot desocupado, y la caja se queda en ese lugar. Codificándolo:

```
        (:action dejar_caja
            :parameters (?r - robot ?obj - caja ?loc - lugar)
            :precondition (and
                (portada ?obj ?r)
                (en ?r ?loc)
                )

            :effect (and
                (en ?obj ?loc)
                (not (ocupado ?r))
                (not (portada ?obj ?r))
                )
        )
```

Definido el fichero de dominio, vamos a crear un problema sencillo donde el robot tenga que mover cuatro cajas hasta un punto de entrega, en la situación inicial tres de las cuatro cajas van a estar en una estantería, la cuarta va estar portada por el robot que se encuentra un punto de partida. Codificándolo en PDDL:

```
; Sección con metainformación:
(define
    (problem fetch_simple_problem)
    (:domain fetch_domain)

    ; Conjunto de objetos:
    (:objects
        fetch - robot
        caja1 caja2 caja3 caja4   - caja
        inicio estanteria entrega - lugar
    )

    ; Estado inicial
    (:init (en fetch inicio)
            (portada caja1 fetch)
            (ocupado fetch)
            (en caja2 estanteria)
            (en caja3 estanteria)
            (en caja4 estanteria)
    )

    ; Objetivos:
    (:goal  (and
            (en caja1 entrega)
            (en caja2 entrega)
            (en caja3 entrega)
            (en caja4 entrega)
            )
    )
)
```

No es muy difícil ver que la solución a este problema es el siguiente plan:

1. El robot se ha de mover al punto de entrega
2. El robot deja la caja que estaba portando
3. El robot va hasta la estantería donde recoge una de las cajas
4. El robot vuelve al punto de entrega y deja la caja que portaba.
5. El robot repite los pases 3 y 4 para las cajas restantes.

Si resolvemos este problema usando el planificador online de planner cloud, comprobamos que la respuesta es :

Figura 32: Solución del problema de demostración

Una vez planteada esta representación minimalista del problema logístico y visto lo sencillo que es representarlo en PDDL, vamos a aumentar la complejidad añadiendo cualidades de PDDL 2.1 para poder llevar a cabo una representación más real. Entre las cosas que vamos a tener en cuenta están la duración de la batería, la distancia entre los distintos lugares de recogida y de entrega de paquetes, el tiempo que tarda en ejecutarse el plan y la masa de las distintas cajas. Los requerimientos necesarios de PDDL2.1 son los siguientes:

- Numeric Fluents: Las variables numéricas son similares a los predicados, son variables que adquieren un valor numérico y que se pueden asociar con un objeto, con varios objetos o con ninguno. Se pueden utilizar en comparaciones para formular precondiciones de acciones, y al igual que pueden mantenerse constante a lo largo del plan puede ser modificadas por las acciones del agente, permitiendo incrementar su valor, decrementar su valor o asignar un valor concreto. También se puede operar con ellas pudiendo realizar sumas,restas,multiplicaciones y divisiones, además se pueden utilizar como métricas para optimizar el plan, indicando si se lo que se quiere maximizar o minimizar una variable numérica durante el plan.

- Durative actions: Una acción prolongada en el tiempo es una acción que requiere una cantidad de tiempo para completarse. La cantidad de tiempo se puede expresar como un valor o como una desigualdad. De manera similar a las acciones tradicionales, tenemos condiciones, en vez de precondiciones, ya que ahora se pueden representar condiciones que existen al principio antes de realizar la acción, condiciones que existan mientras se realiza la acción y condiciones al final de la acción, de la misma manera se pueden representar los efectos en estos tres momentos. Esta diferencia permite a su vez que existan acciones concurrentes, permitiendo una mayor expresividad. Un ejemplo de estas condiciones sería dentro de la planficación de vuelos de aviones, la acción vuelo podría tener como requisito que las pistas estuvieran libres al principio de la acción, en el despegue, y al final de la acción, en el aterrizaje, permitiendo que la pista se encuentre ocupada mientras el avión esta volando.

Empezaremos creando las variables numéricas, como queremos tener en cuenta en el plan la batearía del robot vamos a crear dos variables, la variable batería actual y la variable máxima capacidad de batería, ambas asociadas a objetos tipo robot. Para tener en cuenta las restricciones del peso que puede portar el robot crearemos tres variables, una variable máxima carga asociada al robot, que representará la máxima carga que puede soportar el brazo del robot, la variable masa portada también asociada al robot y que indica cuanta carga porta el robot, y la variable masa caja que estará asociada con cada una de las cajas e indicara el peso de las mismas. Por último para tener en cuenta la localización de los distintos lugares se creará una variable velocidad relacionada al robot y una variable distancia asociada entre dos lugares. Al igual que se ha creado la variable distancia también se podrían haber creado dos variables, la variable x y la variable y que representarán las coordenadas de los distintos lugares y tener en cuenta así la distancia entre ellos. Se prefiere la solución con la variable distancia antes que utilizar un sistema de coordenadas, ya que es bastante difícil y poco legible, expresar en PDDL el cálculo de las la distancia euclídea entre dos puntos, ya que se carece de un operador raíz cuadrada, a pesar de que la solución de distancias haga más lenta la definición del estado inicial en el fichero problema. Todas estas variables se representan en la sección de funciones de PDDL.

En lo que respecta a los tipos y predicados crearemos el tipo estación de carga que es de tipo lugar, y eliminaremos el predicado "ocupado"porque la máxima carga se tendrá en cuenta con las variables numéricas. En lo que respecta a las acciones tendremos que crear la acción cargar batería y modificar las acciones que ya existentes:

- cargar la batería: Los parámetros serán el robot que carga la batería y una estación de carga, para que el robot pueda cargar la batería tiene que permanecer durante toda la acción en la estación de carga, la duración sería igual a dividir la carga que le falta a la batería entre la intensidad de carga, y el efecto es que al finalizar la acción la batería se encontrará en su máxima capacidad. Si quisiéramos hacer la representación más realista podríamos crear la acción cargar batería tiempo mínimo, esta acción tendría una duración mínima y su efecto sería incrementar la carga de la batería el producto entre la corriente de carga y el tiempo mínimo. Igualmente también existen otros mecanismos en PDDL como las funciones continuas para representar esta clase de efectos, esta sintaxis se verá en el próximo capitulo.

- moverse: Los parámetros se mantienen igual que en la anterior representación, pero para poder realizar esta acción el robot tiene que estar en el lugar de origen al

comenzar la acción y el robot tiene que tener batería durante toda la acción, incluyendo el comienzo y el final. La duración es igual a la distancia en metros por la velocidad del robot. El efecto es que al principio la carga se reduce en proporción a la distancia recorrida y que al final de la acción el robot esta en el punto destino.

- recoger caja: Los parámetros se mantienen al igual que en la anterior representación, las condiciones son que el robot y la caja se tienen que mantener en la ubicación del objeto durante toda la acción y que no se ha de superar la carga máxima del robot. Los efectos son que al principio de la acción la masa portada por el robot aumenta tanto como la masa de la caja, y que al final de la acción la caja deja de estar en la ubicación donde se encontraba y es portada por el robot.
- dejar caja: Los parámetros se mantienen al igual que en la anterior representación, las condiciones son que el robot y la caja se tienen que mantener en la ubicación del objeto durante toda la acción. Los efectos consisten en que al final de la acción la masa portada por el robot disminuye tanto como la masa de la caja, y que la caja deja de estar portada por el robot para estar en la ubicación donde se encuentra el robot.

A continuación se muestra la representación de dos de las acciones de este dominio:

```
(:durative-action recoger_caja
    :parameters (?r - robot ?obj - caja ?loc -lugar)
    :duration (= ?duration 10)
    :condition (and
        (at start (en ?r ?loc))
        (over all (en ?r   ?loc))
        (at start (en ?obj ?loc))
        (at start  (<= (masa_portada ?r) (maxima_carga ?r) ))
        (over all  (<= (masa_portada ?r) (maxima_carga ?r) ))
    )
    :effect (and
        (at start (increase (masa_portada  ?r) (masa_caja ?obj)))
        (at end (not (en ?obj ?loc)))
        (at end (portada ?obj ?r))
    )
)

(:durative-action cargar_bateria_entera
    :parameters (?r - robot ?loc - estacion_carga)
    :duration (= ?duration
            ( / (- (max_capacidad_bateria ?r) (bateria_actual ?r))
                (intensidad_carga ?r ))
        )
    :condition (and
        (at start (en ?r ?loc))
        (over all (en ?r ?loc))
    )
    :effect (and
            (at end (assign (bateria_actual ?r) (max_capacidad_bateria ?r)))
    )
)
```

4.4. Creación de comportamientos de alto nivel

Como hemos podido ver, gracias a la abstracción de PDDL hemos sido capaces de generar toda la estructura de planificación sin necesidad de conocer como es la arquitectura real del robot, y como se van a ejecutar las distintas acciones. En este apartado vamos a explicar de forma breve como con la ayuda de los paquetes mencionados al principio, vamos a ser capaces de programar de forma rápida las rutinas de alto nivel que representan las acciones.

- Moverse: Esta rutina es de las más sencillas de implementar, crearemos un nodo en python que utilice el servicio de acción move_base presente en el nav_stack, esta acción se encargará de mover el robot hasta las coordenadas del mapa que se manden.
- Recoger caja y dejar caja: Para las acciones recoger caja y dejar caja crearemos un nodo para cada una. Este nodo va a seguir los siguientes pasos, orientaremos la cabeza del robot para buscar el código presente en la estantería, si no se encontrase el código la rutina habría fallado, en caso de haber sido encontrada podremos calcular sus coordenadas respecto a la cámara gracias al paquete ar_track_alvar, y mediante el uso de transformadas podremos calcular la posición de la estantería respecto al eje de coordenadas de la base del robot. Por último utilizando la interfaz del paquete move_it recogeremos la caja o la dejaremos en su sitio.
- Cargar batería: La acción cargar batería no se representará fielmente a la realidad, puesto que los fabricantes no recomiendan bajo ni circunstancia el proceso de carga sin la manipulación de un humano. Para representarla el robot permanecerá quieto encima de la estación de carga tanto tiempo como la acción requiera.

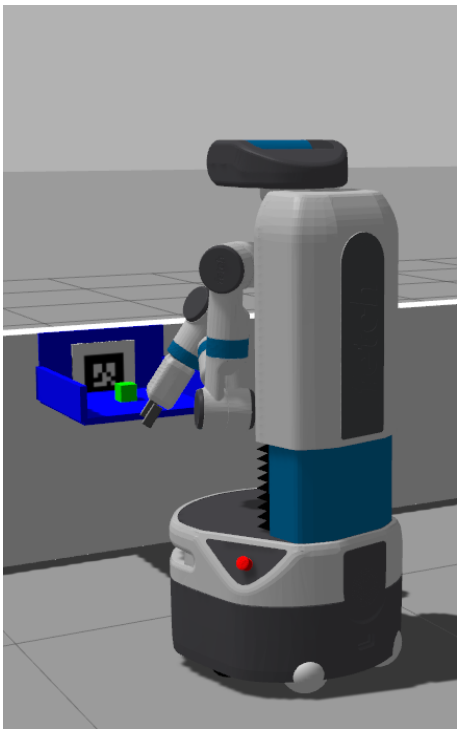

Figura 33: Fetch antes de recoger una caja.

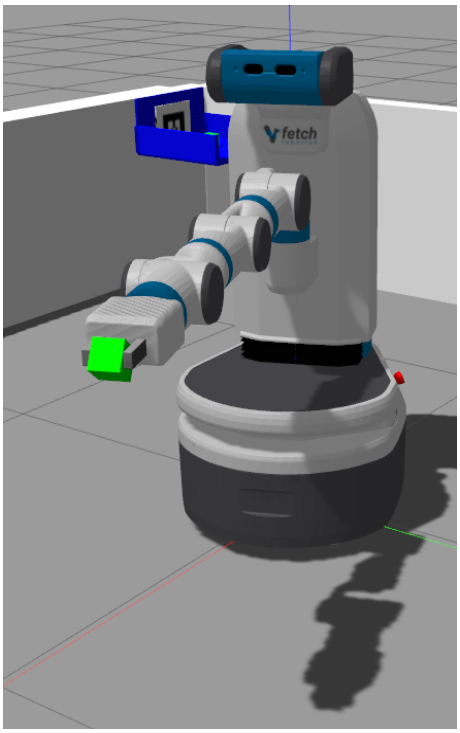

Figura 34: Fetch portando una caja mientras se mueve.

4.5. Integración en Rosplan

Una vez diseñada la representación del problema y las rutinas de alto nivel, solo falta integrarlo en el sistema de planificación. Como hemos visto en el capitulo anterior la arquitectura de Rosplan se integra de forma bastante directa, en la gran mayoría nodos no hay muchos parámetros que configurar ni muchas opciones que estudiar, por lo que lo más complejo es crear las distintas interfaces que habrá entre el sistema de planificación y el resto del sistema.

Empezaremos creando las distintas interfaces para cada acción presentes en el problema de planificación, estas interfaces las podremos crear de dos maneras, utilizando la interfaz genérica que trae consigo Rosplan o creando una desde cero. La ventaja de utilizar la interfaz genérica es que automatiza toda la comunicación con el framework de Rosplan, desde la subscripción a los distintos topics, como la comprobación de los de que los operadores en la acción son correctos, y la actualización de la base de conocimientos. La desventaja es que hay acciones que no se adecuan a este protocolo tan automatizado, un ejemplo son aquellas acciones que no interesa que actualicen la base de conocimientos porque ya hay una interfaz sensorial que se encarga de actualizarla.

En el caso de las acciones coger caja y deja caja crearemos la interfaz utilizando la clase genérica de Rosplan, para ello crearemos una clase en C++ que herede de la clase que contiene la interfaz genérica. Como en el apartado anterior hemos programado las rutinas de alto nivel en Python, tenemos tres opciones; reprogramar todo el código hecho en Python a C++, importar el la clase de Rosplan de C++ a Pyhton o aprovecharnos del sistema distribuido de ROS y crear un nodo con la rutina de alto nivel en Python y otro nodo en C++ que se encargue de la conexión con Rosplan. La opción más correcta y cómoda, puesto que estamos trabajando en ROS, es la tercera ya que nos evita que acoplemos la rutina de alto nivel a Rosplan y la diferencia entre tener un nodo o dos nodos a penas tiene relevancia en el coste computacional, ya que son nodos pasivos que se mantienen todo el rato a la espera de que les llegue una petición, además de que nos permite crear una interfaz con la rutina de alto nivel desde la que podemos acceder desde todos los puntos del sistema.

La interfaz la crearemos utilizando servicios de ROS, aunque también sería buena opción hacerlo mediante acciones de ROS. La petición del servicio sera una cadena con el nombre de la caja que se ha de recoger o dejar, y la respuesta sera un booleano que indique si la rutina de alto nivel ha tenido éxito o no. En lo que respecta a la nueva clase que hereda de la interfaz, crearemos un atributo que almacene el cliente encargado de contactar con el nodo que que implementa la rutina de alto nivel, y modificaremos la función de concreteCallback para hacer la petición a este nodo. La función concreteCallback es una función virtual heredada de la interfaz genérica, esta se ejecuta cuando el nodo dispatch publica la acción asignada a este nodo, esta función devuelve un booleano que indica el éxito de la acción, si la acción tiene éxito se actualiza la base de conocimientos y se sigue con el plan, en caso de no tenerlo la petición se repetirá hasta que el plan se cancele.

Por otro lado la interfaz para la acción de moverse la implementaremos desde cero, esto es debido a que el encargado de actualizar la posición del robot en la base de conocimientos será una interfaz sensorial, y no queremos que se actualice por culpa de la acción moverse. Para crear esta interfaz bastará con suscribirse al topic de action_dispatch, analizar en el callback si la acción corresponde con moverse y si es así anunciar por el topic action_feedback que comienza el proceso de ejecución, en el proceso de ejecución llamará a la rutina de alto nivel y una vez esta haya concluido, solo faltará publicar por el topic action_feedback el éxito o el fracaso de la acción.

En cuanto a la interfaz sensorial para actualizar la posición en la base de conocimientos, se suscribirá al topic que publica la posición en el sistema de navegación, al no tener un sistemas de coordenadas continuo nuestra representación del problema en PDDL, lo que haremos sera buscar cual es el lugar más cercano a la posición real del robot, y actualizar la base de conocimientos indicando que nos encontramos en ese lugar.

Por último crearemos un nodo al que llamaremos planning controler, este nodo se encargará de iniciar el proceso de planificación, llamando a los servicios de los distintos nodos de Rosplan, y hará de interfaz con el usuario para actualizar las base de conocimientos en caso de que sean necesario desplazar más pedidos e iniciar así el proceso de replanificación.

Una vez hecho esto solo faltará crear un launchfile que permita levantar cómodamente todos los nodos involucrados en el sistema, y configure el servidor de parámetros de ROS con los parámetros adecuados. Muchos de estos ya se explicaron en la el capitulo anterior, pero igualmente a continuación se hace un breve resumen de su función y los parámetros configurados:

1. knowledgeBase: El nodo encargado de almacenar la representación PDDL. Aparte de entregar como parámetros la ruta hasta los ficheros PDDL que describen el problema, configuraremos el parámetro use_unknowns como falso, para que aquellos predicados que no se encuentren instanciados en el problema inicial se consideren falsos y no desconocidos. La existencia de este parámetro está definido así para poder aplicar Conformant Planning, un modelo de planificación que permite .

2. problem_interface: Es el encargado de comunicar el estado del problema al resto de nodos y almacenarlo en un fichero PDDL externo que no corresponda con el del usuario.

3. planner_interface: Es el nodo que se encarga de envolver al planificador, dependiendo de la representación que estemos utilizando lo configuraremos par que llame a un planificador o a otro, en el caso de la representación hecha mediante PDDL 2.1 escogeremos POPF, OPTIC, SMT o TFD, ya que admiten todos los requerimientos necesarios para la planificación temporal de la representación, preferiblemente se recomienda usar SMT o Optic puesto que son los más avanzados y versátiles. Si quisiéramos usar la primera representación que es mucho básica se podrían usar MetricFF, ContigentFF o SMT, ya que el dominio utiliza precondiciones negativas, aunque lo más recomendable es usar SMTPlan o ContigentFF por las mismas razones que con la representación anterior.

4. parsing_interface: Se encarga de generar y envolver el plan en una estructurada de datos para que pueda ser fácilmente dividido en acciones y ejecutado . Lo configuraremos para que realice una representación mediante planes secuenciales puesto que no hay acciones concurrentes dentro del dominio.

5. dispatch_interface: Es el encargado de administrar la ejecución del plan.

6. Interfaces coger_caja, dejar_caja y cargar_batería: Son los tres nodos encargados de las interfaces de acción de las rutinas de alto nivel que utilizan la interfaz genérica, al usar la interfaz genérica es necesario configurar el parámetro pddl_action_name con el nombre de la acción que van a ejecutar.

7. Move Action: Es el nodo encargado de ejecutar la rutina de movimiento, contiene su propia interfaz con Rosplan.

8. Sensing Interface: Es la interfaz sensorial encargado de actualizar en la base de conocimientos la posición del robot. Es necesario configurar en los parámetros el fichero yaml de la interfaz sensorial.

9. Planning Controller: Se encarga de iniciar el proceso de planificación en el arranque del sistema, y actualizar la base de conocimientos en función de los paquetes que sean necesarios trasladar.

Una vez acabado el desarrollo del sistema de planificación, somos capaces de crear un launch file que incluya todos los componentes del sistema y ponerlo en práctica. Si levantamos el sistema entero y nos fijamos en el acoplamiento del sistema vemos como los componentes del sistema empiezan a comunicarse entre si. Si nos centramos en el sistema de planificación y analizamos desde el terminal vemos:

1. La interfaz del problema genera fichero PDDL con la instancia actual del problema y comunicándoselo a la planner_interface por el topic /rosplan_problem_interface /problem_instance.

```
lesmus@lesmuspc:~$ rostopic echo /rosplan_problem_interface/problem_instance -n 1 -p
%time,field.data
1630533289306015968,(define (problem task)
(:domain pr2_domain)
(:objects
    pr2 - robot
    caja23 caja35 caja11 caja3 - caja
    inicio estanteria23 estanteria35 estanteria11 entrega53 entrega52 entrega76 - lugar
)
(:init
    (en pr2 inicio)
    (en caja23 estanteria23)
    (en caja35 estanteria35)
    (en caja11 estanteria11)



)
(:goal (and
    (en caja11 entrega53)
    (en caja23 entrega52)
    (en caja35 entrega76)
))
)
```

Figura 35: Gerneración del problema y transmisión vía topic

2. El planner_interface llama al planificador y transmite el plan por el topic /rosplan_planner_interface/planner_output como una cadena de texto.

```
lesmus@lesmuspc:~$ rostopic echo /rosplan_planner_interface/planner_output -n 1 -p
%time,field.data
1630534304223475933,0: (moverse pr2 inicio estanteria11)  [0.001]
1: (recoger_caja pr2 caja11 estanteria11)  [0.001]
2: (moverse pr2 estanteria11 estanteria23)  [0.001]
3: (moverse pr2 estanteria23 entrega53)  [0.001]
```

```
4: (dejar_caja pr2 caja11 entrega53)  [0.001]
5: (moverse pr2 entrega53 estanteria23)  [0.001]
6: (recoger_caja pr2 caja23 estanteria23)  [0.001]
7: (moverse pr2 estanteria23 estanteria35)  [0.001]
8: (moverse pr2 estanteria35 entrega52)  [0.001]
9: (dejar_caja pr2 caja23 entrega52)  [0.001]
10: (moverse pr2 entrega52 estanteria35)  [0.001]
11: (recoger_caja pr2 caja35 estanteria35)  [0.001]
12: (moverse pr2 estanteria35 entrega76)  [0.001]
13: (dejar_caja pr2 caja35 entrega76)  [0.001]
```

Figura 36: Transmisión del plan vía topic

3. El parse_interface envulve el plan en una estructura de datos que se más cómoda para ser separada en distintas acciones y lo publica en /rosplan_parsing_interface/complete_plan.

```
lesmus@lesmuspc:~$ rostopic echo /rosplan_parsing_interface/complete_plan -n 1
nodes:
  -
    node_type: 2
    node_id: 0
    name: "plan_start"
    action:
      action_id: 0
      plan_id: 0
      name: ''
      parameters: []
      duration: 0.0
      dispatch_time: 0.0
    edges_out: [0, 2, 5, 9, 12, 16, 21, 26, 32, 35, 40, 45, 52, 58, 64]
    edges_in: []
  -
    node_type: 0
    node_id: 1
    name: "moverse_start"
    action:
      action_id: 0
      plan_id: 0
      name: "moverse"
      parameters:
        -
          key: "robot"
          value: "pr2"
        -
          key: "from"
          value: "inicio"
        -
          key: "to"
          value: "estanteria11"
      duration: 0.0010000000475
      dispatch_time: 0.0
    edges_out: [1, 3, 6]
    edges_in: [0]
  -
```

Figura 37: Se envuelve el plan en una estructura y se transmite vía topic

4. El dispatch_interface administra la ejecución y va enviando acciones por el topic /rosplan_plan_dispatcher/action_dispatch a medida que va ejecutando

```
lesmus@lesmuspc:~$ rostopic echo /rosplan_plan_dispatcher/action_dispatch
action_id: 0
plan_id: 0
name: "moverse"
parameters:
  -
    key: "robot"
    value: "pr2"
  -
    key: "from"
    value: "inicio"
  -
    key: "to"
    value: "estanteria11"
duration: 0.0010000000475
dispatch_time: 0.0
---
```

Figura 38: Se transmite la primero orden del plan

5. Las interfaces de acción y las interfaz de sensor van actualizan la base de conocimientos y van transmitiendo el feedback de las acciones. Haciendo que la dispatch_interface continue con la ejecución del plan. En este caso vemos como en el topic encargado del feedback se notifica que se ha realizado la acción moverse, y acto seguido por el dispatch_action se envie la siguiente acción del plan recoger caja.

```
lesmus@lesmuspc:~$ rostopic echo /rosplan_plan_dispatcher/action_feedback
action_id: 0
plan_id: 0
status: 1
information: []
---
action_id: 0
plan_id: 0
status: 2
information: []
---
```

Figura 39: Feedback indicando el estado de la acción moverse.

```
---
action_id: 1
plan_id: 0
name: "recoger_caja"
parameters:
  -
    key: "r"
    value: "pr2"
  -
    key: "obj"
    value: "caja11"
  -
    key: "loc"
    value: "estanteria11"
duration: 0.0010000000475
dispatch_time: 1.0
---
```

Figura 40: Finalizada la acción moverse se publica la siguiente acción.

4.6. Técnicas avanzadas

Durante la integración hemos utilizado gran parte de las herramientas que nos brindaba Rosplan, igualmente se puden aplicar técnicas más avanzadas de planificación aprovechando la estructura de ROS.

4.6.1. Portfolio

Muchos planificadores modernos utilizan una técnica conocida como Portfolio o como Plan mergin , consiste en utilizar dentro del mismo planificador varios planificadores con distintas fortalezas y ventajas. Se puede aplicar una versión más simple de esta técnica aprovechando la estructura distribuido de ROS, para ello bastará con lanzar tantos nodos de tipo interfaz de planificación , configurados con distintos planificadores, como planificadores se deseen y será necesario crear un nodo arbitro. Este nodo arbitro lo que tendría que hacer es leer las distintas soluciones al problema que se van publicando en los topics, calcular las distintas variables numéricas y analizar cual es el plan que cumple las mejores métricas, también podría hacer fusiones de planes e incluso podríamos varios planificadores con velocidades muy distintas y en cuanto llegase la primera solución empezar a planificarlo y a medida que fueran apareciendo planes mejores cancelar el plan actual y cambiarlo por el nuevo, si fuesen compatibles en los primeros pasos no sería necesario cancelarlo.

4.6.2. MultiAgent Planning

Aunque en la pagina oficial de Rosplan no aparezcan soportado ninguno planificador para el problema multiagente, hay veces que se puede expresar problemas multiagente en problemas clásicos, de hecho existen programas que transforman los problemas multiagente en representaciones de planificación clásica de manera automática. Esto permite seguir utilizando la estructura de Rosplan para los problemas multiagente y repartir los distintos nodos interfaz de acción entre los distintos robots. Por otro lado hay investigaciones que integran planificación multiagente en Rosplan utilizando planificadores multiagente, para ello lo que hacen es crear su propia interfaz entre el nodo planing_interface y el solver multiagente.

4.6.3. RDDL y PPDDL

La planificación estocástica resulta muy útil en dominios cercanos a la robótica móvil, es por ello que Rosplan extiende la estructura de sus mensajes y de sus interfaces para permitir planificación probabilística, además de crear una interfaz de planificación para el planificador Prost, el campeón IPC probabilística de 2018.

La sintaxis de RDDL esta basada en las redes bayesianas dinámicas, esto permite definir condiciones de acciones mediante funciones de probabilidad, expresar efectos correlacionados, crear variables parametrizadas, permite eventos exógenos, mundos con observabilidad parcial y crear variables enumeradas. En cambio RDDL no permite crear modelos de tiempo continuo, ni acciones prolongadas en el tiempo, las principales características de PDDL 2.1. Igualmente este lenguaje resulta muy útil para representar problemas de navegación, ya que esto permite que el planificador sea capaz de tener en cuenta tanto la posibilidad que haya atascos o incluso accidentes de manera relacional aparte de soportar operaciones que PDDL no soporta como las raíces cuadradas.

5. Integración en entornos robotizados mediante Plansys2

En el capitulo anterior hemos visto como integrar un sistema de planificación en un sistema robotizado utilizando ROS1 y Rosplan. En este apartado volveremos a integrar un sistema de planificación pero utilizando ROS2 y Rlansys2, como muchos de los conceptos y estructura son similares, nos centraremos más concretamente en las ventajas de Plansys2, la representación del problema en PDDL y la nueva versión del framework robótico. En este caso nuestro entorno robotizada va a ser un restaurante, donde los robots además de actuar camareros van a prepara la propia comida. Este entorno nos va a dar la posibilidad de poner en práctica algunas de las extensiones de PDDL más útiles en el mundo de la robótica no habíamos podido ver hasta ahora, y algunas de las carencias del lenguaje de acción.

5.1. Dominios PDDL

Una de las principales ventajas de Plansys2 es que podemos mezclar diversos dominios de PDDL, creando problemas más complejos, lo que permite una estructura mucho más modular y un sistema de planificación más dinámico. Esta opción añadida a que los launch files de ROS2 se escriben en python en vez de XML, tenemos un sistema mucho más flexible y robusto, ya que podemos ver examinar la situación del entorno en launchfile y añadir más o menos ficheros de dominio dependiendo de la situación del entorno. Por ejemplo si en nuestra supuesto restaurante tenemos un tipo especial de robot que dispone de ciertas acciones que el resto de robots no pueden manejar, no es necesario adentrase dentro del sistema, y cambiar el problema PDDL o manipular el launchfile para poner el de repuesto, basta con que la rutina de arranque modifique la cantidad de dominios que se van a utilizar. Aprovechando esta posibilidad crearemos dos dominios distintos, uno para representar el problema que representa la cocina y otro para las acciones de camarero. La versión de PDDL que utilizaremos será la 2.1 haciendo especial incidencia en la planificación temporal y concurrente.

En dominio de la cocina vamos a tener un robot que va a ser capaz de moverse por la cocina, transportar ingredientes, manipular los ingredientes para hacer ingredientes más avanzados, y combinar distintos ingredientes para formar platos que luego se podrán llevar a los consumidores. Las dos primeras acciones, moverse y recoger ingredientes, no son nuevas ya las planteamos en el problema logístico y solo se van a ver ligeramente modificadas. En cambio las dos siguientes son completamente nuevas y presentan un efecto completamente distinto al resto de acciones que ya hemos visto, mezclar ingredientes o refinarlos tienen como principal efecto la creación de un nuevo elemento y la desaparición de los ingredientes que se han utilizado para conseguirlo, estos efectos, crear y destruir, no se pueden modelar de forma cómoda en ninguna de las versiones oficiales de PDDL, puesto que no se puede modificar el número de objetos definidos en la situación inicial.

Esta carencia para representar esta clase de efectos lleva siendo conocida aproximadamente desde la IPC de 2002, donde se quiso recrear un problema que representara una fabrica de automóviles y hubo problemas para definir la acción fabricar un coche. Desde entonces se han planteado multitud de soluciones para acabar con esta falta de flexibilidad, muchas de estas tienen que ver con la creación de nuevos efectos como new para crear objetos o delete para eliminarlos. Y aunque haya habido peticiones por parte de la comunidad para integrarlos en posteriores versiones de PDDL, esto no se ha realizado porque la introducción de estos efectos conduciría no solo a estados de tamaño variable,

sino también al tamaño variable de acciones. Esto es un desafío en el lado de la implementación, ya que conceptos como la síntesis invariante o el análisis de accesibilidad son significativamente más difíciles de implementar si el número y la forma de las acciones no está claro.

Igualmente parte de la comunidad ha creado herramientas para transformar de manera automática dominios utilizando estos operadores a codificaciones PDDL oficiales, aunque no siempre es posible realizar estas transformaciones sobre todo si se utilizan el operador delete y el forall en conjunto. Es por ello que nosotros utilizaremos una de las técnicas más habituales para poder modelar estos operadores, aunque sea solo un parche ante esta falta de expresividad. La técnica consiste en crear uno o dos predicados que representen la característica de existir y no existir, habitualmente serán necesarios dos predicados debido a la falta de poder utilizar precondiciones negativas en muchos planificadores. Por otro lado esta técnica necesita que todos aquellos objetos que se vayan a crear o a destruir estén definidos en la situación inicial del problema, imponiendo un número máximo de objetos que se pueden crear y destruir. Esta solución también provoca que la planificación conlleve más tiempo, puesto que todos los objetos están presente desde estado inicial, lo que conlleva a que el planificador tenga que revisar todas las acciones posibles sobre objetos y que haya un factor de ramificación enorme, puesto que hay un montón de objetos reserva sobre los que se puede aplicar la misma acción, además de construir un grafo muy difícil de podar si solo se aplica poda detección de ciclos.

Comentados todos estos aspectos sobre como realizar una posible implementación, vamos a crear un dominio sencillo donde lo podamos poner en práctica. Este primera prueba la vamos a realizar sin utilizar acciones prolongadas en el tiempo, puesto que para aplicar la técnica no son necesarias y la sintaxis de las acciones instantáneas es más sencilla, aunque como hemos adelantado el dominio final si tendrá este tipo de acciones. Este primera implementación es bastante directa, la acción amasar necesita que el robot y la harina estén en el mismo lugar, que la harina exista y que la masa que se va a crear no exista. El efecto provoca que la harina no exista y la masa exista, puesto que no disponemos de precondiciones negativas es necesario esta redundancia de predicados en los efectos.

```
(:action amasar
        :parameters (?r - robot ?harina -harina ?masa - masa ?loc - lugar)
        :precondition (and
                (en ?harina ?loc)
                (en ?r      ?loc)
                (existe     ?harina)
                (no_existe  ?masa)
        )

        :effect (and
                (en ?masa ?loc)
                (not (no_existe ?masa))
                (existe ?masa)
                (not (existe ?harina))
                (no_existe ?harina)
        )
)
```

Esta acción es la representa directa de la técnica mencionada antes, pero dado nuestro dominio se pueden realizar ciertas mejoras. Como en nuestro dominio para poder utilizar un ingrediente o realizar cualquier acción es necesario que el robot y el objeto compartan ubicación, existe una manera más eficiente de hacer que un mismo ingrediente no se pueda reutilizar, y es eliminándolo de cualquier lugar. Este cambio permite que sola exista un único objeto base de cada tipo, se entiende por objeto base aquel objeto que no se puede formar mediante ninguna acción, en este caso la harina sería un elemento base pero la masa no. Codificando esta mejora obtenemos:

```
(:action amasar
        :parameters (?r - robot ?harina -harina ?masa - masa ?loc - lugar)
        :precondition (and
                (en ?harina ?loc)
                (en ?r      ?loc)
                (no_existe  ?masa)
            )

        :effect (and
                (en ?masa ?loc)
                (not (no_existe ?masa))
                (existe ?masa)
                (not (en ?harina ?loc))
            )
)
```

En cambio con esta modificación es necesario crear una acción que permite obtener más objetos base y poner un limite a su obtención, ya que antes estas acciones las condicionaba el número de objetos presente en el estado inicial del problema. Modelar esta acción es sencilla, para crear un objeto solo hay que volver a poner el elemento base en el campo de juego, ya sea poniendo el objeto en un lugar o dandoselo al robot para que lo porte, y para generar un limite será necesario contar con variables numéricas. Al estar modelando un restaurante crearemos la acción coger harina de la despensa, en esta acción la variable numérica no esta asociada a ningún objeto opción que habíamos comentado pero no habíamos vista ahora.

```
(:action coger_harina_de_la_despensa
        :parameters (?r - robot ?harina -harina ?loc - despensa)
        :precondition (and
                (en ?r      ?loc)
                (> (harina_disponible) 0)
            )

        :effect (and
                (portada ?harina ?r)
                (decrease (harina_disponible) 1)
            )
        )
)
```

Para comparar ambas implementaciones vamos a plantear un problema de planificación donde se tengan que crear n masas, para resolver este problema utilizaremos el online planner de la planning wiki. En la situación inicial todas los objetos base estarán en la misma despensa, permitiendo que la primera implementación solo tenga repetir n veces la acción de amasar, en cambio la segunda tendrá que repetir n veces el ciclo de coger ingrediente de las despensa, dejar ingrediente y amasar.

Número de objetos	Primera implementación	Segundo implementación
5	0.268 s	0.263 s
25	0.663 s	0.408 s
50	1.939 s	0.839 s
100	Error	2.26 s

Tabla 1: Comparación entre ambos dominios

Como se puede ver la diferencia con pocos objetos no es amplia, pero a medida que se van necesitando más objetos de tipo base la diferencia empieza a aumentar considerablemente debido al factor de ramificación. Tal es la diferencia, que cuando son necesarios 100 objetos el solver es incapaz de manejar la memoria necesaria para mantener todas las ramas, y lanza una excepción en el programa.

Por último vamos a transformar estas acciones en acciones prolongadas en el tiempo, a diferencia del dominio logístico este dominio nos va a permitir hacer planificación con acciones concurrentes ya que acciones como cocinar en el horno algunos ingredientes no necesitan que el robot se halle delante del horno, si no que puede encontrar realizando otras acciones mientras tanto. A continuación se presentan dos de las acciones presentes en el dominio final, como podemos ver ambas podrían transcurrir de forma concurrente permitiendo crear planes más eficientes:

```
(:durative-action amasar
    :parameters (?r - robot ?harina -harina ?masa - masa ?loc - lugar)
    :duration (= ?duration 3)
    :condition (and
            (at start (en ?harina ?loc))
            (at start (en ?r      ?loc))
            (at start (no_existe  ?masa))
            (over all (en ?r ?loc))
            (over all (en ?harina ?loc))
            (over all (no_existe  ?masa))

        )

    :effect (and
            (at end (en ?masa ?loc))
            (at end (existe ?masa))
            (at end (not (no_existe ?masa)))
            (at end (not (en ?harina ?loc)))
        )
)
```

```
(:durative-action hornear
    :parameters (?r - robot ?horno - horno ?plato - plato)
    :duration (= ?duration 30)
    :condition (and
        (at start (en ?r   ?loc))
        (at start (en ?obj ?loc))
        (at start (and (>= (temperartura ?horno) 200) (<= (temperartura ?horno) 220) )
        (over all (and (>= (temperartura ?horno) 200) (<= (temperartura ?horno) 220) )
        (over all (en ?obj ?horno))
        (at end   (en ?r ?horno))
    )

    :effect (and
        (at end (horneado ?plato))
        (at end (portada   ?plato ?robot))
        )
)
```

Por otro lado en el dominio del problema camarero el robot va a tener en cuenta el grado de satisfacción de los comensales, para tener esto en cuenta crearemos una variable numérica que sea la satisfacción de cada uno de nuestros comensales, esta satisfacción aumentará o decrecerá en función del tiempo que tarde el robot en servirles y atenderles, y será la métrica que el planificador optimice durante la creación del plan. Es un ejemplo algo simple, pero nos servirá para darnos cuenta de la necesidad de poder crear eventos exógenos.

Para empezar tendremos que incluir el requerimiento continuos-efects, los efectos continuos permiten modelar los efectos como funciones numéricas relacionadas con el tiempo que se tarda en ejecutar una acción. Este requerimiento ya se menciono como una solución al problema de como representar la carga de la batería en el capitulo anterior, en su momento planteamos hacer una función carga máxima que cargará la batería hasta al máximo, puesto que el fetch necesita apoyo humano para cargarse, y sería molesto pedirle ayuda cada muy poco tiempo. Con los efectos continuos podemos dejar que el planificador lleva a cabo un razonamiento más elevado sobre esta acción, ya que dejaremos la duración del tiempo de carga a decisión del planificador. Codificando esta nueva versión:

```
    (:durative-action cargar_bateria
        :parameters (?r - robot ?loc - estacion_carga)
        :duration ()

        :condition (and
            (at start (en ?r ?loc))
            (over all (en ?r ?loc))
        )
        :effect (and
                ((increase (bateria_actual ?r) (*  (intensidad_carga ?r ) #t)))
    )
)
```

Con las acciones durativas podríamos intentar expresar la acción de esperar por parte de un comensal, consistiría en que si un comensal ha pedido nota y por tanto esta esperando su plato, su felicidad baja mientras dure la acción. Esto que parece una implementación correcta, no lo es, puesto que el planificador no va a realizarla jamás debido a que es una acción, y el planificador tiene libertad para escoger las acciones que forman dentro del plan, así que jamás la va a escoger porque bajaría la calidad del plan. Si quisiéramos que esta acción funcionase tal y como esta codificada necesitaríamos eventos, los eventos son acciones que se ejecutan una vez las condiciones son alcanzadas independientemente de la decisión del planificador. El problema es que los eventos son una característica bastante avanzada de PDDL y ninguno de los planificadores compatibles con Plansys2 lo toleran. Es por tanto que la solución clásica para poder expresar este tipo de acciones, donde el planificador no obtiene ninguna ventaja , es metiendo estos efectos dentro de una acción más grande que tenga que realizar obligatoriamente para finalizar la tarea. Esto perjudica sobre todo a la legibilidad del dominio y a crear acciones que no tienen del todo sentido, pero no afectan al rendimiento como el anterior inconveniente.

5.2. Integración de rutinas de alto nivel

Al igual que con Rosplan, una vez realizada la descripción del problema toca crear las interfaces entre el nodo que administra el plan y los nodos que ejecutan las rutinas de alto nivel. A diferencia del sistema de planificación de ROS1 donde estas interfaces se podían realizar utilizando una clase genérica de C++, que automatizaba todos la comunicación, o desde cero, donde el usuario utilizando la mensajería de ROS creaba su propia interfaz, en este framework las interfaces se tienen que realizar obligatoriamente en C++ utilizando una clase genérica, puesto que en vez de utilizar ROS para la comunicación utiliza un protocolo interno del framework.

Igualmente la forma de utilizar ambas interfaces genéricas no es muy distinta, en ambas la forma más sensata de integrarla es creando una segunda clase que herede de la presentada por el framework y modificar la función de callback. Esta función de callback es la función que se ejecuta cuando el nodo tiene que realizar la rutina de alto nivel, en el caso de Plansys2 esta función se llama do_work y es de tipo void en vez de booleana. Pero aunque la forma de utilizarla sea parecida, existen ciertas mejoras en las del más moderno:

1. Realimentación: Es posible comunicar al nodo encargado de administrar el plan, cual es el estado actual de la ejecución de la rutina en cada momento, para ello existen dos canales de comunicación, el de feedback y el de succes. El canal de feedback permite mandar el porcentaje que se lleva realizado de las acción, junto a una cadena de texto que permita conocer al usuario el estado interno de la acción. Por otro lado el canal de succes permite indicar el éxito o fracaso de la acción, además de una cadena de texto que representa el estado interno del nodo.

2. CascadeLifecycleNode: En ROS2 existe el concepto de ciclo de vida de un nodo, el ciclo de vida no es más que una maquina de estados propia del nodo que representa su estado interno, los cuatros estados principales son Unconfigured, Inactive, Active y Finalized. Dependiendo en que estado esta el nodo, el nodo puede realizar unas cosas u otras, por ejemplo un nodo inactivo no es capaz de leer por topics, ni es capaz de atender a peticiones de servicio o acciones, haciendo que fallen automáticamente todas las peticiones. Este concepto permite que el usuario tenga un mayor control

sobre el estado del sistema de ROS y es relevante porque la interfaz de Plansys2 internamente es un nodo, por lo que tiene esta maquina de estados interna y se puede utilizar para crear comportamientos más complejos. Además esta interfaz es un tipo especial de nodo llamado nodo cascada o en serie, los nodos cascada son capaces de activar a otros nodos cascada cuando se encuentran en estado activo, este comportamiento es útil cuando una acción requiere activar un nodo que procesa la información de un sensor, ya que el nodo encargado del sensor solo estará activo, mientras la acción que requiere la información este lo esté.

Por otro lado Plansys2 nos ofrece una manera más modular para agrupar las distintas interfaces, Plansys2 pone a disposición del usuario una forma bastante eficiente para crear arboles de comportamiento con la librería BeheaviurTree.cpp, pudiendo representar cada una de las acciones del dominio PDDL con un árbol de comportamiento. La construcción de estos arboles se hacen mediante la combinación de distintos tipos de nodos, los nodos se clasifican en los siguientes tipos:

- Nodos de control: Los nodos de control son nodos que tienen uno o varios nodos hijos, cuando este tipo de nodo recibe la señal de ejecución, lo que hace es propagarla al resto de los hijos. Un ejemplo de un nodo de control es el nodo sequence, este nodo lo que hace es propagar la señal de ejecución al primer hijo, si el hijo devuelve una señal de éxito el nodo sequence manda la señal de ejecución al siguiente hijo, en cambio si el hijo devuelve una señal de fracaso el nodo padre devuelve una señal de fracaso. En caso de que todos los hijos devuelvan la señal de éxito el nodo sequence propaga la señal de éxito hacia arriba.
- Decoradores: Los decoradores cumplen la misma función que los nodos de control pero solo tienen un hijo. Uno de los decoradores más importantes es el nodo retry, este nodo propaga la señal de ejecución a su nodo hijo, hasta que este devuelve una señal de éxito o se supera el número de llamadas máximo que han sido configuradas en el nodo padre.
- Nodos de acción: Los nodos de acción corresponden con las hojas del árbol de comportamientos, por lo que no tienen hijos. Es el usuario el que tiene definir el comportamiento de estos nodos cuando llega una señal de ejecución. Dentro de los nodos de acción existe una variante especial llamada nodos de condición, estos nodos están obligados a tener un comportamiento síncrono, por lo que las únicas señales que puede devolver son la señal de éxito o la de fracaso. En nuestro caso este tipo de nodos lo ocuparan los nodos de ROS encargados de las rutinas de más bajo nivel.

Para definir una acción de PDDL como un árbol de comportamiento e integrar la interfaz de Plansys2 son necesarias dos cosas, un fichero XML en el que se describa los nodos que conforman el árbol y su estructura. Y crear los nodos de acción que compondrán el árbol, estos nodos de acción se construyen de manera similar a la interfaz anterior pero en este caso en vez de usar un CascadeLifecycleNode utiliza un BtActionNode, lo que permite a demás de interactuar con el sistema de ROS2 interactuar con la librería BeheaviurTree.cpp.

La decisión sobre si utilizar una única interfaz o un árbol de comportamiento, depende de lo complejo o simple que sea el sistema para realizar la rutina de alto nivel, o la cantidad de los elementos que se vayan a reutilizar entre unas rutinas u otras. Hacer un árbol para una acción que requiere un solo nodo como sería la acción moverse, que normalmente se

define como la petición desde un nodo al sistema de navegación, no tiene mucha sentido. En cambio si para realizar una rutina es necesario la coordinación secuencial de múltiples nodos, como podría ser coger un objeto con la ayuda de un sistema de visión o transportar un objeto desde una coordenada a otra si que podría ser útil.

Otro de las grandes ventajas de Plansys2 sobre Rosplan es la identificación de flujos de acciones durante el plan, esta optimización permite que varios agentes puedan realizar un plan secuencial e incluso la misma tarea, lo que acelera la ejecución del plan. Aunque este ejecución con varios agentes pueda parecer planificación multi-agente sigue sin serlo, puesto que no están presente características en el desarrollo del plan como la coordinación entre agentes o el concepto de privacidad.

6. Aplicación en competiciones de robótica

Durante el desarrollo de este Trabajo Fin de Grado se han aplicado algunas de las técnicas presentadas en un sistema robotiza real. Se ha integrado un sistema de planificación en la arquitectura software de los dos robots autónomos que ha diseñado el equipo UAH Robotics Team. Estos robots móviles son capaces de realizar las distintas tareas presentes de la edición de Eurobot 2021.

6.1. Eurobot 2021 Sail The World

Eurobot es una competición internacional de robótica creada en 1998 dedicada a universidades y clubes de robótica. En esta competición equipos formados por estudiantes con menos de 25 años participan diseñando y construyendo uno o dos robots autónomos móviles de dimensiones acotadas, el objetivo de estos es jugar partidos contra los robots del resto de equipos.

Los partidos consisten en un periodo de 100 segundos, donde los robots tienen que realizar ciertas tareas dentro de un campo conocido, cada vez que realizan una tarea de forma correcta su equipo consigue puntos, ganando el partido el que más puntos tenga. Todos los años tanto el campo como las tareas cambian puediendo encontrar la normativa en la página web oficial de Eurobot [31], este año la temática es la navegación marítima por lo que el tablero de juego esta conformado en las siguientes zonas de carácter marítimo:

1. Zona de inicio.
2. Puertos marítimos.
3. Veleta.
4. Líneas de calle.
5. Mangas de viento.
6. Arrecifes o dispensadores.
7. Zona de amarre.
8. Anclaje de faro.

Además en el suelo del campo se encuentran distintos vasos que el robot puede recoger.

Figura 41: Campo de Eurobot 2021

Las tareas con las que el robot puede conseguir puntos son las siguientes:

- Se han de recoger los distintos vasos y llevarlos hasta la zona de los puertos marítimos (2). Estos vasos están repartidos tanto en el suelo del campo como en los dispensadores (6), los vasos pueden ser de dos colores rojos o verdes, si se colocan los vasos en las lineas de calle (4) de su respectivo color se consiguen puntos extra.

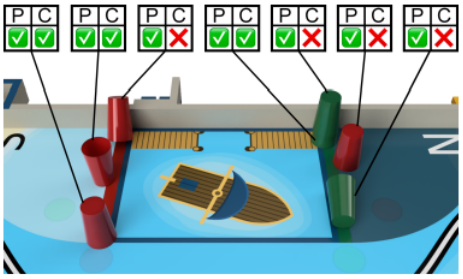

Figura 42: Almacenar las boyas en los canales

- Levantar las mangas de viento: Las mangas de viento (5) tienen un punto de apoyo sobre el que se pueden rotar, si se rotan estas banderas el equipo gana puntos.

Figura 43: Mangas de viento

- Encender el faro: Cada equipo puede colocar un robot estático en la zona de faro (8), este robot estático tiene que elevarse en medio del partido hasta alcanzar una altura determinada, este robot tiene que ser activado en medio del partido por alguno de los robot móviles teniendo que haber un contacto físico entre ellos.
- Anclar con seguridad: Si al finalizar el partido el robot se encuentra parado en uno de las zonas de amarre (7) este gana puntos. Además al inicio del partido la veleta (3) empezará a rotar, si al final del partido el robot se encuentra en zona de amarre que marca la veleta al final de su movimiento, norte o sur, se obtendrán más puntos.

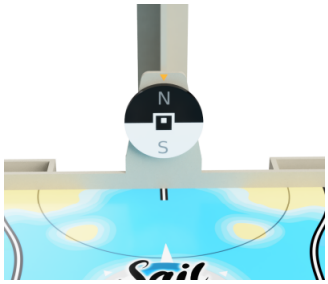

Figura 44: Brújula

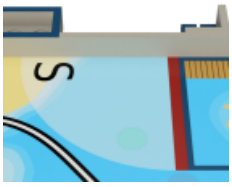

Figura 45: Zona de amarre Sur

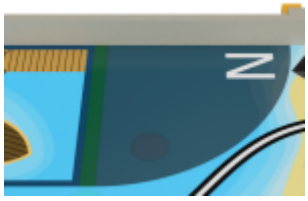

Figura 46: Zona de amarre Norte

- Izar la bandera: Al final del juego, entre el segundo 95 y el 100, los robots podrán alzar una bandera que lleven equipada por encima suya, consiguiendo puntos.

6.2. Robots

Para realizar estas tareas el UAH Robotics Team ha diseñado los siguientes robots:

El robot de dispensadores es el encargado de activar el faro y recoger los vasos que se encuentran en los dispensadores. Para ello cuenta con un brazo de dos grados de libertad en su estructura modular, este brazo tiene como herramienta final un array de 5 ventosas conectadas a un sistema neumático, lo que le permite recoger los vasos y desplazarlos hasta el puerto. Para más información sobre el diseño de este robot, se pueden consultar los trabajos de fin de grado de Daniel Blanco [2] y Carlos Mamblona [24].

El robot de suelo es el encargado de tirar las mangas de viento, izar las banderas y cogerlos vasos que se encuentran a nivel del suelo. Para ello en su diseño integra dos compartimentos donde almacenar los vasos que va recogiendo del suelo, además de contar con dos servo motores que permiten tanto tirar las mangas de viento, y un mecanismo piñón cremallera que le permite izar la bandera.

El robot estático cuenta con un sistema de elevación basado en un mecanismo de doble tijera, lo que le permitirá pasar desde la altura mínima hasta la máxima manteniendo la estabilidad. Para que uno de los robots móviles pueda activar el mecanismo es suficiente con que se acerque y presionen el botón central, al presionar este botón centrar el mecanismo interno, que cuenta con un sistema de amortiguación, presionaría los dos finales de carrera sin desplazar la estructura.

Figura 47: Robot de dispensadores

Figura 48: Robot de suelo

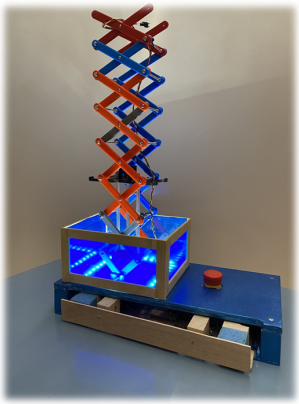

Figura 49: Robot estático

6.3. Estructura Software

La estructura Software esta basada en el uso de ROS Noetic, para ello se utiliza Ubuntu Server 20.04 como sistema operativo y una Raspberry Pi 3 Model B como microordenador. Además cada uno de los robots se utilizan dos microcontroladores de la familia Arduino, que sirven como driver con el sistema de tracción y con el brazo. Para comunicar la Raspberry Pi con los microcontroladores se establece una comunicación mediante el puerto serie entre los distintos dispositivos, ocupando la raspberry el rol de maestro y las arduinos el de esclavo.

En lo que respecta a arquitectura del sistema de ROS encontramos una versión de la arquitectura jerarquizada presentada en el capitulo 4, esta arquitectura es la misma para los dos robots pero parte del funcionamiento de alguno de los nodos es distinta. Las capas que lo conforman son:

- Drivers: Se encargan de comunicarse con los sensores y actuadores. En el caso de los nodos brazo y tracción se comunican con los microcontroladores.
- Navegación y Percepción: Se encargan de indicar a los driver los pasos necesarios para realizar el plan, además de analizar la información entregada por los sensores. Al tener ambos robots actuadores relativamente simples, se elimina el nodo encargado de la planificación de trayectorias de actuadores.
- Rutinas: Realiza la función de interfaz con el sistema de planificación y administra la ejecución del plan.
- Sistema de planificación: Calcula las acciones utilizando Rosplan como framework.
- Administrador del sistema: Es el encargado de levantar los nodos, administrar su funcionamiento y hacer de interfaz con el usuario.

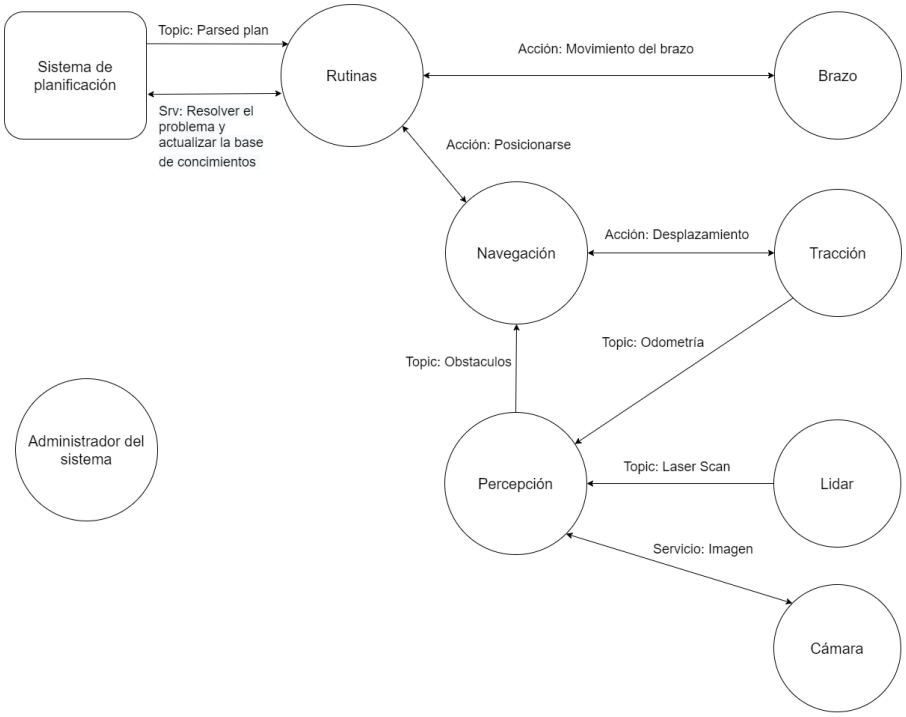

Figura 50: Arquitectura de ROS UAHR 2021

6.4. Representación del dominio

La representación PDDL se realizo mediante el uso de PDDL2.1, ya que como se ha visto a lo largo del trabajo es lo suficientemente flexible para representar dominios robotizados, y es compatible con la gran mayoría de planificadores adaptados a Rosplan y a Plansys2. Para la representación se crearon dos dominios, uno para cada robot, esto es debido a que dentro del campo de juego existen muchas interferencias, lo que dificulta la comunicación entre robots y la posibilidad de desarrollar planificación multiagente o un sistema de planificación centralizado.

Entre los requerimientos que se utilizan destaca la ausencia de las acciones prolongadas en el tiempo a cambio de dar un mayor uso a las variables numéricas. Este cambio respecto a dominios anteriores es debido a la existencia de un único agente y a la falta de acciones concurrentes que pueda realizar el robot, a cambio se creará una variable numérica tiempo para representar el transcurso temporal sobre la que el planificador intentará optimizar el plan de manera directa o indirecta. Un ejemplo de este tipo de acción es la acción soltar vasos propia del dominio del robot de dispensadores:

```
(:action soltar_vasos
    :parameters(?robot - robot ?puerto - puerto)
    :precondition (and
        (en ?robot ?puerto)
        (lleno ?robot)
    )
    :effect (and
        (not (lleno ?robot))
        (increase (puntos)         14)
        (increase (tiempo_partido) 20)
    )
)
```

Por otro lado una de las mayores dificultades a la hora de expresar este problema en PDDL es expresar el objetivo de los robot y el fin de un partido. Si se analiza el objetivo del robot de forma vaga, se ve que a primera vista el objetivo del robot es ganar, este es un objetivo bastante difícil de expresar, podríamos intentar expresarlo como conseguir más puntos que el equipo enemigo, pero al comienzo del partido los puntos de los dos equipos son cero por lo que el planificador no se tendría que esforzar demasiado. Si en cambio expresáramos ganar como conseguir todos los puntos posible, esto llevaría a que el planificador fallase en medio del partido al ver que no es capaz de conseguir todos los antes de que se acabase el tiempo del partido. La solución a este problema es más sencilla, se expresará que el objetivo del robot sea acabar el partido. Esto provocará que el planificador no falle en medio del partido ya que el partido acaba independientemente de las acciones que realice robot. Por otro lado para asegurar que el robot realice puntos, bastará con pedirle al planificador que optimice el plan utilizando alguna relación con la variable puntos. Durante el desarrollo del proyecto se han probado a optimizar dos relaciones:

- Maximizar la variable de puntos.
- Maximizar la densidad de puntos, es decir, el cociente entre los puntos que se consiguen y el tiempo necesario para conseguirlos.

En el dominio de competición se optimizará la función de densidad de puntos, debido a que existen fallos en el sistema de navegación que provocan el bloqueo del robot en medio del partido, impidiendo que el robot se mueva y siga haciendo puntos. Obteniendo mejores resultados al hacer puntos rápidos, en vez de realizar jugadas lentas con grandes beneficios. Codificando estos objetivos en PDDL obtenemos:

```
(:goal (and (fin game)))
(:metric maximize (/ (puntos) (tiempo_partido)))
```

Por otro lado la dificultad de acabar el partido radica en la falta de eventos exógenos, lo que provoca que acabar el partido tienen que ser implementado mediante una acción, al igual que pasaba con el dominio del camarero presentado en el capitulo anterior. Al ser una acción el planificador tiene derecho a decidir cuando aplicarla, lo que provocaría que dependiendo de como se expresase el objetivo se querría realizar en todo momento o se evitaría su aplicación. En caso de expresar el objetivo del problema como en el párrafo anterior, veríamos que el planificador solucionaría el plan con una única acción que acabase el partido. Es por ello que para evitar la constante aplicación de esta acción, la precondición debe de estar ligada a la variable que representa el tiempo de partido. Esta condición ha de expresarse en forma de rango temporal, puesto que si se expresase como una condición en un segundo fijo por ejemplo el segundo 100, que coincide con el tiempo del partido, obligaría al planificador a conseguir una secuencia perfecta de acciones que acabase exactamente en este segundo, cosa que es bastante difícil debido a que la acción de moverse del robot no tienen porque tener una duración entera. Es por ello que lo que hay que hacer es expresar esta condición en forma de rango, para que el planificador pueda tener una mayor flexibilidad a la hora de encontrar un plan. Sigue siendo ligeramente una peor solución que la de los eventos, ya que obliga a crear una acción extra lo que aumenta ligeramente el coste computacional y cede al planificador la decisión de elegir si el partido se acaba o no, cosa que dependiendo del objetivo del problema no es del todo útil. Codificando esta acción en PDDL:

```
(:action fin_partido
    :parameters (?game - partido)
    :precondition (and
        (>= (tiempo_partido) 98)
        (<= (tiempo_partido) 100)
    )
    :effect (and
        (fin ?game)
    )
)
```

Respecto a los dominios presentados en capítulos anteriores se mantienen muchas de las técnicas ya aplicadas, como utilizar un sistema de distancias entre las distintas localizaciones en vez de un sistema de coordenadas, o crear una acción que incrementa ligeramente una variable numérica para representar efectos relacionados con el tiempo, como se hacía en la primera representación de cargar batería en el capitulo cuatro. La ausencia de utilizar efectos continuos es debido a que son incompatibles si no se utilizan acciones prolongadas en el tiempo.

Finalmente para representar esta situación de manera detallada y expresar el complejo comportamiento del partido, son necesarias:

- Los siguientes doce objetos: Lugar, Robot, Partido, Objetivo, Puerto, Zona de amarre, Veleta, Manga de viento, Vaso, Bandera, Faro y Dispensador. Formando el siguiente árbol de herencias:

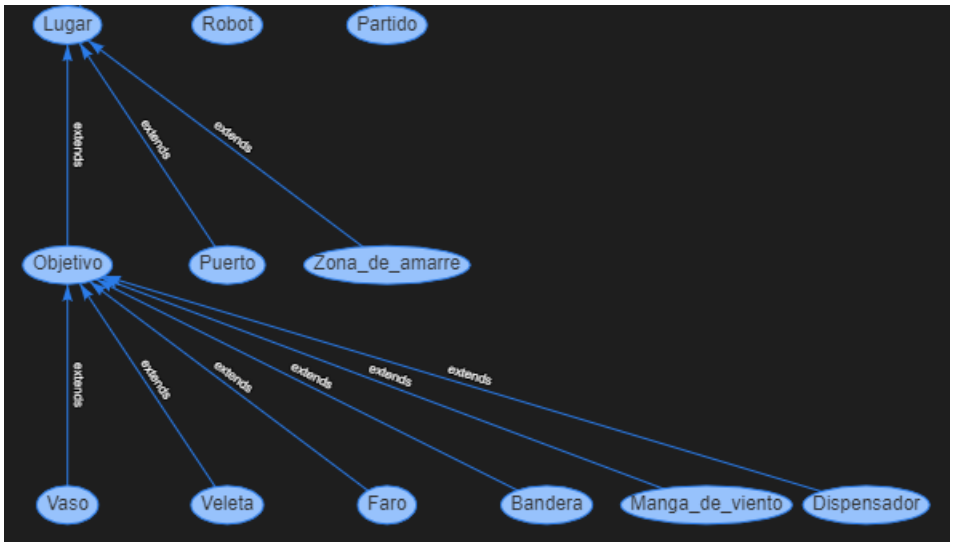

Figura 51: Árbol de herencias entre objetos PDDL

- Los siguientes seis predicados:
  - En: Relaciona la ubicación de un robot con un lugar, indica en que lugar se encuentra el robot.
  - Disponible: Esta relacionado con un único objetivo, indica si la tarea sigue pudiendo realizarse, por ejemplo cuando se activa el faro como no se puede activar más de una vez deja de estar disponible.
  - Lleno: Esta relacionado con el robot de dispensadores, indica si el robot tiene las ventosas ocupadas impidiéndole recoger más vasos de los dispensadores.
  - Color: Esta relacionado con cada uno de los vasos e indica cual es su color. Es interesante ya que existe un bonus por cada parejas de vasos de distinto color que se deje en las lineas de calle, lo que provoca que el robot no solo busque recoger los vasos más cercanos para dejarlos cuanto antes, si no que también intente realizar parejas de colores.
  - Aparcado: Esta relacionado con los robots, indica que el robot esta aparcado y por tanto no se puede mover.
  - Fin: Esta relacionado con el partido e indica el fin de este.
- Las siguientes 6 funciones:
  - Tiempo de partido: No esta asociado a ningún objeto e indica los segundos transcurridos de partido.
  - Distancia: Cumple la misma función que en los dominios de las secciones anteriores, indica la distancia que hay entre dos lugares.
  - Puntos: No esta asociado a ningún objeto e indica los puntos obtenidos al realizar las distintas acciones.

- Vasos verdes y vasos rojos almacenados: Son dos funciones distintas asociadas al robot de suelo, indica cuantos vasos tiene almacenados de cada color, permiten al robot calcular cuantos puntos obtiene al dejar los vasos y si es capaz de almacenar algún vaso más en el deposito.
- Puntos estratégicos de un dispensador: Esta asociado a un dispensador, permite modificar al usuario el comportamiento del robot de dispensadores ya que provoca que el planificador piense que ciertos dispensadores ofrecen más puntos de lo habitual. Es útil porque permite incentivar al robot a realizar jugadas más defensivas o mas agresivas como recoger el dispensador que se encuentra cerca de la zona de inicio del equipo enemigo.

- 16 acciones, estas acciones se pueden clasificar dependiendo del dominio en el que aparecen:
  - En dominio del robot de dispensadores son necesarias 11 acciones de las cuales 4 son únicas estas son: recoger dispensador, soltar vasos de forma precisa, soltar los vasos rápido y activar experimento. Se crean dos acciones de soltar vasos para representar una en la que dejan los vasos de cada color en su linea de calle, y otra donde se sueltan todos los vasos dentro del puerto marítimo independientemente del color, siendo esta segunda acción un movimiento mucho más rápido.
  - En el dominio del robot de suelo son necesarias 12 acciones de las cuales 5 son únicas, estas son tirar las manga, subir la bandera , almacenar un vaso rojo, almacenar un vaso verde y dejar los vasos.
  - Las acciones que comparten ambos robots son moverse, mirar la veleta, esperar hasta que acabe el partido, aparcar en la zona de amarre marcada por la veleta, aparcar en en la zona de amarre por defecto y finalizar el juego.

A continuación se muestra una de las acciones del dominio del robot de dispensadores, siendo un buen ejemplo de la estructura que comparten la gran mayoría de las acciones, esta estructura consiste en:

- Precondiciones: La tarea que se quiere realizar tiene que estar disponible, el robot tiene que estar en el lugar correspondiente y habitualmente cada acción tiene una precondición adicional. En este caso como el robot estático tarda como máximo en subir 10 segundos tiene que ser activado antes del segundo 90.
- Efectos: El objetivo deja de estar disponible y se incrementan tanto los puntos como el tiempo del partido.

```
(:action activar_faro
    :parameters (?robot - robot ?faro - faro)
    :precondition (and (en ?robot ?faro)
        (disponible ?faro)
        (< (tiempo_partido) 90)
    )
    :effect (and (not (disponible ?faro))
        (increase (puntos)         15)
        (increase (tiempo_partido)  5)
    )
)
```

6.5. Integración

Para integrar este sistema de planificación no utilizaremos la metodología estándar que se ha estado utilizando en los capítulos anteriores, en vez de utilizar toda la estructura del framework de planificación descartaremos ciertas partes de la estructura del sistema. En nuestro caso omitiremos tanto la parte del dispatch plan, como las interfaces de acción y las interfaces sensoriales. A cambio la ejecución de las distintas rutinas de alto nivel van a almacenarse y centralizarse en el nodo rutinas, siendo este el que administre la ejecución del plan en vez del nodo dispatch plan, y también se encargara de la actualización de la base conocimientos, en vez de realizarlo las interfaces de acción y las interfaces sensoriales.

Que el nodo rutinas se encargue de tantas cosas es debido a dos razones:

- Tener un mayor control sobre el sistema antes de cada partido y sobre el comportamiento del robot en medio del partido.
- Evitar cálculos o comunicaciones que no sean de provecho dentro de la estructura.

Estas dos razones tienen un mayor peso en este sistema debido a la diferencia con los sistemas robotizados presentados en los apartados anteriores, donde el periodo de funcionamiento del sistema era largo y el funcionamiento general era el mismo durante casi todos los periodos. En este sistema de competición el periodo de funcionamiento es corto y muchas veces es importante modificar el comportamiento antes de cada partido, pudiendo inyectar planes secuenciales fijos a través de un pendrive antes de cada partido o fijar comportamientos en un punto determinado del partido, como intentar bloquear al robot enemigo en un segundo determinado. Además al ser pocas las acciones las que puede realizar el robot y disponer de poco tiempo para recalcular, tiene poco sentido que se estuviese actualizando constantemente la base de conocimientos, además de que el planificador replanificase cada pocos segundos de manera automática. Por otro lado cuando se estuvieron realizando pruebas la raspberry-py a veces no tenía suficiente fuerza de computo para mantener las técnicas de visión artificial, el procesamiento de los datos del sensor lidar, mantener la comunicación con el puerto serie, intentar actualizar el mapa del entorno y actualizar la base conocimientos constantemente, por lo que se opto por una arquitectura más simple y centralizada que permitía reducir el coste computacional.

El funcionamiento del nodo rutinas se pude simplificar en los siguiente pasos:

1. Si no existe un plan inicial indicado por el equipo, llama al servicio generate_problem para que el planificador disponga del problema ,acto seguido llama al servicio planning_server para resolverlo y finalmente ejecuta el servicio parse_plan para facilitar la lectura.
2. Lee el plan por el topic y lo almacena en una lista.
3. Consume la rutina correspondiente del plan, para ello ejecuta una maquina estados asociada con la rutina de alto nivel. En esta maquina de estados se establecerá comunicación con los nodo de navegación y brazo, que serán los que indican a los drivers como ejecutar las rutinas.
4. Si la rutina ha concluido correctamente, se analizará cuanto tiempo a tardado y si se ha deteriorado la calidad del plan, en caso de que el plan se siga ajustando a una cuota fitness se continuará ejecutando, por lo que volverá a repetir este mismo paso. En caso de no ajustarse a la esta cuota o no haberse consumido correctamente el nodo continuará al siguiente paso.

5. Se observan los valores que han sido modificados en el campo respecto a la representación presente en la base de conocimientos, y se actualiza la base de conocimientos mediante el servicio rosplan_knowledge_base/update_array.

6. Se realizan las mismas peticiones al sistema de planificación que se habían realizado en el paso 1 y se continua el flujo del programa en el paso 2.

6.6. Conclusiones del proyecto

En las pasadas ediciones de Eurobot el comportamiento del sistema había sido programado utilizando maquinas de estado finitas, arboles de comportamiento e incluso en la edición de 2020 se intento llevar a cabo un sistema de planificación al observar sus ventajas frente al resto de sistemas. El sistema de planificación anterior utilizaba un único nodo escrito en python que se encargaba tanto de la representación del problema como de su resolución, para resolverlo utilizaba un algoritmo de búsqueda en profundidad y para representar el problema empleaba la formalización proposicional, utilizando una única estructura para representar el conjunto de proposiciones, y funciones que modificaban la estructura para representar las acciones. En comparación con el sistema de este año el nodo de python del año anteriores se siente como una versión tosca y primitiva frente a las ventajas del sistema de planificación de PDDL, siendo mucho más rápida y eficiente la resolución mediante un planificador, y siendo una estructura más modular y sencilla la representación mediante este lenguaje, convirtiendo en las dos únicas desventaja la expresividad que permite la sintaxis de python frente a la de PDDL y que Rosplan resulta ser un sistema bastante complejo de implementar si no se tiene cierta experiencia con ROS.

En lo que respecta a los próximos año, seguramente el equipo siga manteniendo la planificación automática mediante Rosplan como herramienta para definir el comportamiento del robot, ya que ha mostrado grandes resultados durante las pruebas de este año superando con creces a todos los si temas anteriores, además de no estar previsto cambiar la arquitectura software del equipo a ROS2 por el momento. Sin embargo la opción de expresar las rutinas de alto nivel mediante arboles de comportamiento de Plansys2 lo convierte en una opción más que interesante, además de contar con todas las ventajas del nuevo framework robótico.

Por otro lado, gracias a este sistema se han conseguido resultados notables en las competiciones de este año, como ser los segundos clasificados en el campeonato nacional de Eurobot España, lo que habilito que el equipo pudiese participar en la competición internacional de Eurobot en Francia. Se decidió no participar en la competición internacional por los riesgo sanitarios debidos al Covid-19. Igualmente dentro de la estructura Software del robot había diversos fallos que a veces impedían su correcto funcionamiento, como el sistema de navegación que provocaba que en ocasiones el robot se bloquease o el modulo de la visión artificial que impedía coger los vasos correctamente.

7. Conclusiones

Como hemos visto los sistemas de planificación automática son una herramienta más que útil con la que modelar el comportamiento de un robot, llegando a ser bastante sencillo y rápido de implementar en un sistema robotizado. Ya que en parte se pueden realizar descripciones del entornos de forma rápida con PDDL, y representar el entorno con un grado de detalle más que suficiente. Además, la integración en el sistema robotizado se puede hacer de manera rápida si el robot utiliza ROS, actual estándar de facto.

Igualmente la planificación automática muestra ciertas desventajas frente a sus contrincantes lo que las hace una opción menos llamativa, como la falta de una plataforma o fundación con la que aprender los distintos conceptos, la falta de documentación de los distintos planificadores, y a veces su difícil instalación, lo que lo convierte en un mundo muy difícil de entrar. En parte muchos de estos de estos inconvenientes son debido a que la gran mayoría de usuarios de PDDL son parte de la comunidad científica, es esperable, puesto que PDDL fue un intento de crear un estándar para que los distintos grupos de investigación en el ámbito de la planificación automática, tuvieran un mismo lenguaje con el que comparar resultados, pero esto ha provocado que parte de la industria no conozca estos instrumentos. Igualmente parte de la comunidad de planificación esta haciendo grandes esfuerzos por acabar con estos inconvenientes, han creado la planning wiki, una plataforma con la que aprender los distintas conceptos básicos y con información sobre muchas de las herramientas existentes. También han creado el Solver.Planning.Domain, un servidor web en el que se encuentra un planificador y se pueden enviar ficheros PDDL para que los resuelva, evitando el complejo proceso de instalación de algunos planificadores.

Respecto a las paquetes de ROS, se nota como Rosplan fue ideado por expertos en la planificación automática, pero hay ciertos puntos de la estructura de ROS que podrían ser mejorados, la estructura está algo sobrecargada y no utiliza las acciones de ROS, que serian de gran ayuda en algunos puntos de la estructura de planificación. Por otro lado la última versión de ROS compatible con el paquete fue ROS Melodic y no hay indicios de que quieran trasladar el paquete a una nueva versión de ROS. En cambio Plansys2 se encuentra en buena forma, han anunciada ya el desarrollo para la nueva distribución de ROS2, Galactic Geochelone, a pesar de haberse estabilizado recientemente en la versión Foxy Fitzroy. Y aunque es un paquete que aun sigue en desarrollo, tiene muchas características útiles para la robótica que superan a las de su antecesor, aun así se echan en falta conceptos como las interfaces sensoriales, o un mayor número de plugins con los que utilizar otros planificadores, ya que actualmente solo se encuentran dos y con solo características de planificación temporal.

Sobre las bases de este trabajo se pueden desarrollar muchas investigaciones relacionadas, puesto que se trabaja en dos campos muy amplios. Pero algunos de los más cercanos son:

- Crear una interfaz de planificación para Rosplan con un planificador hasta ahora no compatible.
- Crear un plugin compatible con Plansys2 que envuelva un planificador hasta ahora no compatible.
- Desarrollar un planificador PDDL.
- Integrar RDDL en una sistema robotizado real utilizando Rosplan para ello.

A. Estructura y presupuesto del proyecto

En el caso de querer adaptar un sistema de planificación similar al presentado en este Trabajo fin de grado a un proyecto de carácter industrial, se necesitarían realizar ciertas inversiones y una estructuración del proyecto para poder llevarlo a cabo.

Como se ha visto durante la sección 4 la sección 5 e incluso la sección 6, donde se integraban sistemas de planificación a distintos sistemas robotizados, por regla general se cumple una misma arquitectura software en casi todos los sistemas robotizados, esta es la arquitectura jerarquizada presentada en el apartado 4.2. La existencia de esta arquitectura permite organizar el proyecto bajo la siguiente estructura genérica, evidentemente siempre se pueden realizarse modificaciones respecto a la estructura que se muestra a continuación:

1. Adaptación de la estructura software del robot a ROS: Las herramientas utilizadas en en este TFG necesitan construirse sobre una arquitectura basada en ROS, por lo que es obligatorio disponer de esta arquitectura. Al ser ROS un estándar de Facto, sobre todo en la robótica móvil, tal vez este paso no fuese necesario dependiendo del proyecto. Igualmente si fuese necesaria la adaptación no tendría porque rehacerse toda la estructura software del sistema, ROS cuenta con una serie de herramientas ideadas para este tipo de ocasiones llamadas Rosbridges, estas herramientas ponen una API a disposición del usuario con la que comunicarse desde programas no hechos en ROS con los nodos de ROS.

2. Creación de las rutinas de alto nivel: Dependiendo de como este implementado el comportamiento del sistema robotizado, más concretamente los planificadores dependientes del dominio y la integración sensorial, la creación de estas rutinas podrían ser más complicadas o menos complicadas, generando una gran variación en la duración de esta fase. En el caso de que la estructura del sistema trabaje utilizando correctamente la filosofía de ROS, como se vio en el capitulo 4 con el robot movil Fetch, no debería de ser una fase compleja.

3. Creación de los ficheros encargados de representar el problema: Es una de las partes centrales del sistema de planificación, su duración es completamente dependiente de la complejidad del sistema y el nivel de detalle con el que se quiera trabajar. Igualmente como hemos visto en el capitulo anterior no debería de ser muy duradera en el tiempo.

4. Búsqueda del planificador de dominio más eficiente: Existen multitud de planificadores que pueden resolver problemas de PDDL, teniendo cada uno sus ventajas y sus desventajas. Una gran forma de obtener una idea del rendimiento de un planificador es mirando la puntuación obtenida en la IPC, igualmente no todos los planificadores disponibles han participado en la IPC o son más eficientes para un problema concreto. Por lo que es recomendable que se establezca una fase en la que probar los distintos solvers que permiten Rosplan y Plansys2, y ver cual es el más adecuado para el dominio diseñado.

5. Creación de los nodos interfaz entre el framework de planificación y el resto de la estructura software: Puede ser uno de los pasos más laboriosos a la hora de crear el sistema de planificación, no tanto por las interfaces con las rutinas de nivel alto que deberían de haberse creado en el paso dos, si no por como actualizar la base de conocimientos de manera consistente utilizando los sensores, además de ser necesaria la creación de una interfaz entre el planificador y el usuario.

Si se realiza un análisis del coste temporal del proyecto se observa:

Fase	Duración media
Adaptación de la estructura software del robot a ROS	112 Horas
Creación de las rutinas de alto nivel	90 Horas
Creación de los ficheros encargados de representar el problema	40 Horas
Búsqueda del planificador de dominio más eficiente	80 Horas
Creación de los nodos interfaz	160 Horas
Total	482 Horas

Tabla 2: Planificación temporal

Las fases que se presentan arriba se encuentran ordenadas de tal manera que se adapta primera la estructura, se construye la capa intermedia y después se crear el sistema de planificación. Pero como se ha visto una de las ventajas de PDDL es la capacidad de realizar una representación sin saber como se van a realizar las rutinas de alto nivel, por lo que si tuviésemos un equipo conformado por varias personas se podrían organizar el proyecto con el siguiente diagrama PERT:

Fase	Actividad
Adaptación de la estructura software del robot a ROS	A
Creación de las rutinas de alto nivel	C
Creación de los ficheros encargados de representar el problema	B
Búsqueda del planificador de dominio más eficiente	D
Creación de los nodos interfaz	E

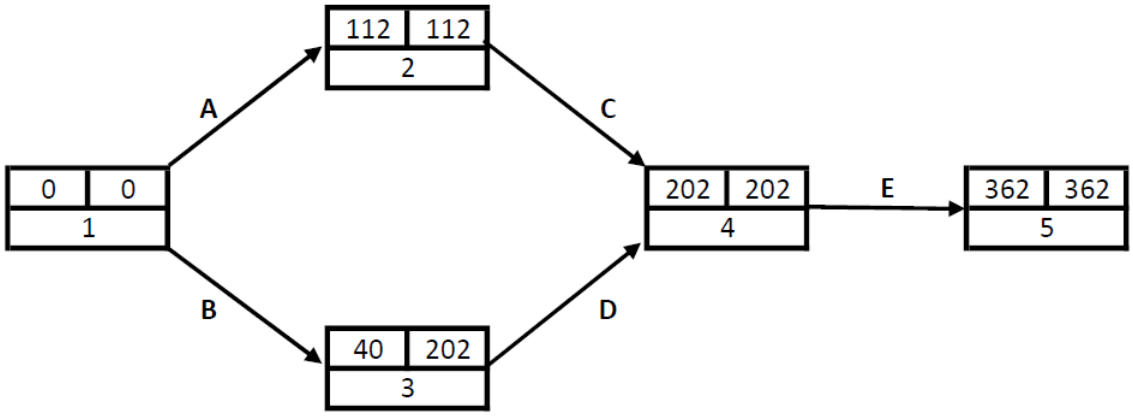

Figura 52: Diagrama PERT del proyecto

Se puede apreciar que el camino critico tiene una duración de 362 horas y lo conforman la adaptación de la estructura software del robot a ROS, la creación de las rutinas de alto nivel y la creación de los nodos interfaz. Justo este camino es el más dependiente de la arquitectura software en la que se va a integrar el sistema de planificación, por lo que además de ser el más largo es el que más incertidumbre posee.

A continuación se desglosan en distintos apartados la inversión necesaria:

Entre las inversiones hardware iniciales se necesitarían principalmente un ordenador con el que desarrollar el sistema de planificación y el sistema robotizado donde integrarlo. El ordenador tienen que contar con las prestaciones necesarias para la utilización del software requerido, como se menciono en el capitulo del estado del arte, ROS esta diseñado para no tener un elevado coste computacional por lo que se pueden utilizar para el diseño del sistema ordenadores de gama media-baja, o incluso microordenadores como la Raspberry-Pi o la Orange-Pi. Sin embargo las simulaciones realizadas en Gazebo son algo más costosas, por lo que seria necesario optar por un ordenador de gama media, aunque estas simulaciones no fuesen necesarias para el desarrollo del sistema son bastante convenientes y forman parte intrínseca de la filosofía de ROS como se ha visto en el trabajo.

Concepto	**Articulo**	**Coste**
Ordenador	Ordenador ASUS i7	512 €

Tabla 3: Costes Hardware del proyecto

Respecto a las herramientas software podemos ver como el uso de tecnología open source permite un abaratamiento de los costes, consiguiendo un coste nulo en estas herramientas. En la siguiente tabla se muestran dos posibles combinaciones, la primera de ellas utiliza ROS1 con Rosplan y la segunda utiliza ROS2 con PlanSys2.

Concepto	**Articulo**	**Coste**
Sistema Operativo	Ubuntu 18.04	0 €
ROS	Melodic Morenia	0 €
Framework de planificación	Rosplan	0€

Concepto	**Articulo**	**Coste**
Sistema Operativo	Ubuntu 20.04	0 €
ROS	Foxy Fitzroy	0 €
Framework de planificación	Plansys2	0 €

Tabla 4: Costes de herramientas Software del proyecto

Por último para adaptar el sistema de planificación sería necesario contratar a uno o varios ingenieros encargados de adaptar la estructura y su programación. Considerando que la contratación de ingenieros supone un coste uniforme por hora:

Concepto	**Sueldo**	**Horas totales**	**Coste**
Ingeniero	13.5 €/hora	482 horas	6507 €

Tabla 5: Costes de la mano de obra del proyecto

Pudiendo resumir la inversión total necesaria:

Concepto	**Coste**
Coste Hardware	512 €
Coste Software	0 €
Coste Humano	6507 €
Coste Total	7019 €

Tabla 6: Costes total del proyecto

Referencias

[1] Ángelal BARROSO PEÑA. "Estudio de la importancia del modelado en planificación automática". En: (2016).

[2] Daniel Blanco Martín. "Diseño de un actuador de agarre para EUROBOT 2020". En: (ene. de 2021).

[3] Jorge Brea y col. "A Study for the Application of Automated Planning to Mobile Assistive Robots". En: *Cybernetics and Systems* 45 (ago. de 2014). DOI: 10.1080/01969722.2014.945310.

[4] Gerard Canal y col. "Probabilistic Planning for Robotics with ROSPlan". En: *Towards Autonomous Robotic Systems*. Ed. por Kaspar Althoefer, Jelizaveta Konstantinova y Ketao Zhang. Cham: Springer International Publishing, 2019, págs. 236-250. ISBN: 978-3-030-23807-0.

[5] Michael Cashmore y col. "Rosplan: Planning in the robot operating system". En: *Proceedings International Conference on Automated Planning and Scheduling, ICAPS* 2015 (ene. de 2015), págs. 333-341.

[6] Wenjun Cheng y Yuhui Gao. "Using PDDL to Solve Vehicle Routing Problems". En: *Intelligent Information Processing VII*. Ed. por Zhongzhi Shi y col. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, págs. 207-215. ISBN: 978-3-662-44980-6.

[7] Planning Comunity. *Planning Wiki*. URL: https://planning.wiki/.

[8] João Fabro y col. "ROS Navigation: Concepts and Tutorial". En: vol. 625. Feb. de 2016, págs. 121-160. ISBN: 978-3-319-26052-5. DOI: 10.1007/978-3-319-26054-9_6.

[9] Open Source Robotics Foundation. *Ros Foxy Documentation*. URL: https://docs.ros.org/en/foxy/index.html.

[10] Open Source Robotics Foundation. *Ros Nav Stack DOcumentation*. URL: http://wiki.ros.org/navigation.

[11] Open Source Robotics Foundation. *Ros Webpage*. URL: http://wiki.ros.org/.

[12] Maria Fox y Derek Long. "PDDL2.1: An extension to PDDL for expressing temporal planning domains". En: *J. Artif. Intell. Res. (JAIR)* 20 (dic. de 2003), págs. 61-124. DOI: 10.1613/jair.1129.

[13] Cristóbal Tapia García. "Diseño e implementación de un planificador para un agente autónomo". 2017. URL: http://oa.upm.es/46988/.

[14] Alfonso Gerevini y Derek Long. "Plan Constraints and Preferences in PDDL3 The Language of the Fifth International Planning Competition". En: *ICAPS 2006* (ene. de 2005).

[15] M. Ghallab, D. Nau y P. Traverso. *Automated Planning: Theory and Practice*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 2004. ISBN: 9781558608566. URL: https://books.google.es/books?id=eCj3cKC%5C_3ikC.

[16] Malik Ghallab, Dana Nau y Paolo Traverso. *Automated Planning and Acting*. Jul. de 2016. ISBN: 9781107037274. DOI: 10.1017/CBO9781139583923.

[17] Malik Ghallab y col. "PDDL - The Planning Domain Definition Language". En: (ago. de 1998).

[18] J. Hoffmann. "The Metric-FF Planning System: Translating". En: (jun. de 2011).

[19] Yu-qian Jiang y col. "Task planning in robotics: an empirical comparison of PDDL- and ASP-based systems". En: *Frontiers of Information Technology Electronic Engineering* 20 (mar. de 2019), págs. 363-373. DOI: 10.1631/FITEE.1800514.

[20] Lentin Joseph. "Programming with ROS". En: mayo de 2018, págs. 171-236. ISBN: 978-1-4842-3404-4. DOI: 10.1007/978-1-4842-3405-1_5.

[21] KCL. *RosplanWeb*. URL: https://kcl-planning.github.io/ROSPlan/.

[22] David V. Lu. *map$_s$erverpackagedocumentation*. URL: http://wiki.ros.org/map_server.

[23] Nerea Luis, Susana Fernández y Daniel Borrajo. "Plan merging by reuse for multi-agent planning". En: *Applied Intelligence* 50.2 (feb. de 2020), págs. 365-396. ISSN: 1573-7497. DOI: 10.1007/s10489-019-01429-0. URL: https://doi.org/10.1007/s10489-019-01429-0.

[24] Carlos Mamblona Nieto. *Diseño estructural y sistema de tracción diferencial de un robot autónomo de competición para Eurobot 2020 y 2021*. Ene. de 2021.

[25] Francisco Martín y col. *Optimized Execution of PDDL Plans using Behavior Trees*. Ene. de 2021.

[26] Francisco Martın y col. "PlanSys2: A Planning System Framework for ROS2". En: *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - October 1, 2021*. IEEE, 2021.

[27] Scott Niekum. *ar$_t$rack$_a$lvarpackagedocumentation*. URL: http://wiki.ros.org/ar_track_alvar.

[28] E. Pednault. "ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus". En: *KR*. 1989.

[29] Poom Pianpak, Tran Son y Zachary Toups. "A Multi-agent Simulator Environment Based on the Robot Operating System for Human-Robot Interaction Applications: 21st International Conference, Tokyo, Japan, October 29-November 2, 2018, Proceedings". En: oct. de 2018, págs. 612-620. ISBN: 978-3-030-03097-1. DOI: 10.1007/978-3-030-03098-8_48.

[30] Scott Sanner. "Relational Dynamic Influence Diagram Language (RDDL): Language Description". http://users.cecs.anu.edu.au/ ssanner/IPPC$_2$011/*RDDL.pdf*. 2010.

[31] Planet de Science. *Eurobot Web*. URL: https://www.eurobot.org/ (visitado 13-09-2021).

[32] M. Wise y col. "Fetch & Freight : Standard Platforms for Service Robot Applications". En: 2016.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá